

# Introduction

# Course Overview

- Anatomy of a Supercomputer
- Supercomputing – Empowering Exploration
- Performance – Oriented Theme
  - Where does performance come from?
  - Sources of performance degradation
- Supercomputer System Stack
- A Brief History of HPC
  - Enabling technologies
  - Fundamental Concepts
  - Accomplishments in HPC



# Learning Outcomes

- Define *supercomputer*
- Describe performance Issues in HPC
- Define *scalability*
- List machine parameters affecting performance and describe how they impact operation
- Describe driving factors for HPC
- Describe sources of performance degradation
- Describe supercomputing system stack
- Understand the evolution and drivers of the 7 decade history of HPC

# Biggest Computer in the US - Supercomputer



The TITAN petaflops machine fully deployed at Oak Ridge National Laboratory in 2013. It takes up more than **4000 square feet** and consumes approximately **8 Megawatts** of electrical power. It has a theoretical peak performance of over **27 Petaflops** and delivers 17.6 Petaflops  $R_{\max}$  sustained performance for the HPL (Linpack) benchmark. This architecture includes Nvidia GPU accelerators.

# Addressing the Big Questions

- What grand challenge applications demand these capabilities?
- How do users program such systems?
  - What languages and in what environments?
  - What are the semantics and strategies?
- How to manage supercomputer resources to deliver useful computing capabilities?
  - What are the hardware mechanisms?
  - What are the software policies?
- What are the computational models and algorithms that can map the innate application properties to the physical medium of the machine?
- How to integrate enabling technologies into computing engines?
- How to push the performance to extremes?
  - What are the enabling conditions?
  - What are the inhibiting factors?

# Goals of the Course

- A first overview of the entire field of HPC
- Basic concepts that govern the capability and effectiveness of supercomputers
- Techniques and methods for applying HPC systems
- Tools and environments that facilitate effective application of supercomputers
- Hands-on experience with widely used systems and software
- Performance measurement methods, benchmarks, and metrics
- Practical real-world knowledge about the HPC community
- Accessible by students outside the HPC mainstream

# Student Objectives

- Computational Scientists
  - How to use supercomputer
  - Apply to domain applications
- HPC research
  - Foundations of supercomputing
  - Research directions
- System Administration
  - Making them serve the user base
  - Managing multi-billion dollar systems
- Design Engineering
  - Challenges to building supercomputers
  - What are their enabling technologies
  - Functional components



# Course Overview: Major Topic Areas

- Introduction to Supercomputer Systems, Supercomputing, and Technologies
- Distributed Memory Systems and MPI Programming
- Shared Memory Systems and OpenMP Programming
- Performance Modeling and Measurement for Performance Optimization
- User Environments: System Software and Tools
- Enhanced Techniques for System Use
- Conclusions and Future Directions

# Course Overview: First Pass Introduction

- An Overview
  - What's it all about
  - How you can take advantage of it
- Parallel Computer Architecture
  - What are supercomputers
- Commodity Clusters
  - The most widely used form of HPC systems
  - So easy you can build one yourself; many do
- Benchmarking
  - Finding out how good your system is compared to others
- Throughput Computing
  - The easiest way to use HPC systems
  - “Embarrassingly parallel”

# Course Overview: Distributed Memory and MPI Programming

- Communicating Sequential Processes (CSP)
  - Abstract model of computing across distributed
- Enabling Technologies – Networks for Node Integration
  - Making a big supercomputer out of smaller computers
  - Communications, data movement, and messaging
- MPI Programming
  - Most widely used programming interface
  - True scalable computing
  - Suitable for current generation supercomputers
- Performance Measurement
  - What do we want to make better
  - Units of measure for system operation
  - Some methods for instrumentation

# Course Overview: Shared Memory Systems and OpenMP Programming

- Single Node Architecture
  - Every thing you need to compute in one box
- Enabling Technologies – Memory, Core Architectures,..
  - Basic component types and their underlying technologies
- Parallel Thread Computing
  - The abstract execution model for shared memory multiple threads
- OpenMP programming
  - Widely used parallel programming interface
  - Incremental changes to sequential applications
- Performance factors and measurement
  - Parameters that determine the capabilities of shared memory systems

# Course Overview: System Software

- Operating Systems
  - Low level software for hardware resource management
  - Memory management and address allocation
  - Processor core schedulers for application threads and processes
  - I/O interfaces
- Schedulers and Middleware
  - Medium level software that controls system wide resources
  - Controls user jobs and total system workload
  - Provides system administration tools
- Parallel File I/O
  - Supports persistent storage for disks and tapes (sometimes NVRAM)
  - Exploits multiple disks for higher bandwidth and reliability

# Course Overview: Advanced Methods

- **Visualization**
  - Software to render raw output data in a form that is visually informative
  - Many distinct types of visual data presentation in 2-D and 3-D
- **Parallel Algorithms**
  - Abstract patterns of computation that achieve desired computation results (correct answers) and exposes and exploits parallelism
- **HPC Libraries**
  - Prepackaged functions that already provide frequently used algorithms
  - Simplifies and expedites creation of new HPC applications
- **Heterogeneous Computing and GPUs**
  - Systems comprising different types of functional units
  - In some cases accelerates computations compared to homogeneous

# Course Overview: Final Topics and Future Directions for you and HPC

- What's beyond the scope of this course?
- What form will the future of HPC take?

# Recent #1 Supercomputers



Tianhe-1A  
2.56 Pflops



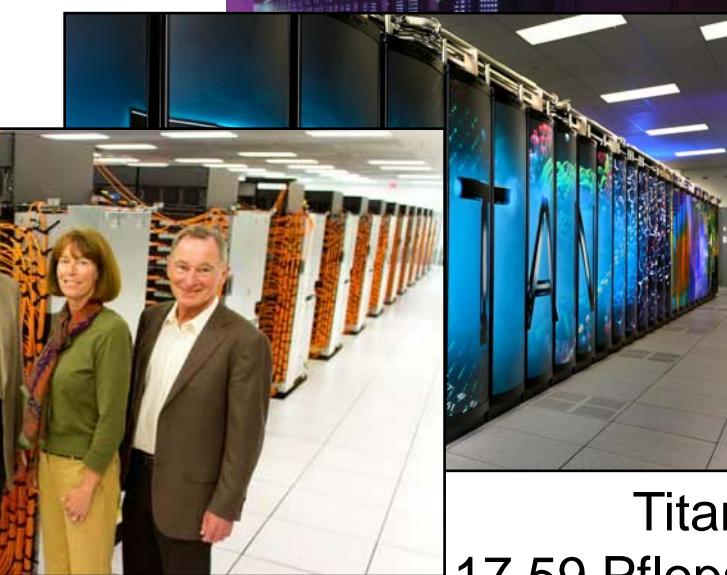
Kei  
10.5 Pflops



Sequoia  
16.32 Pflops

Kei

10.5 Pflops

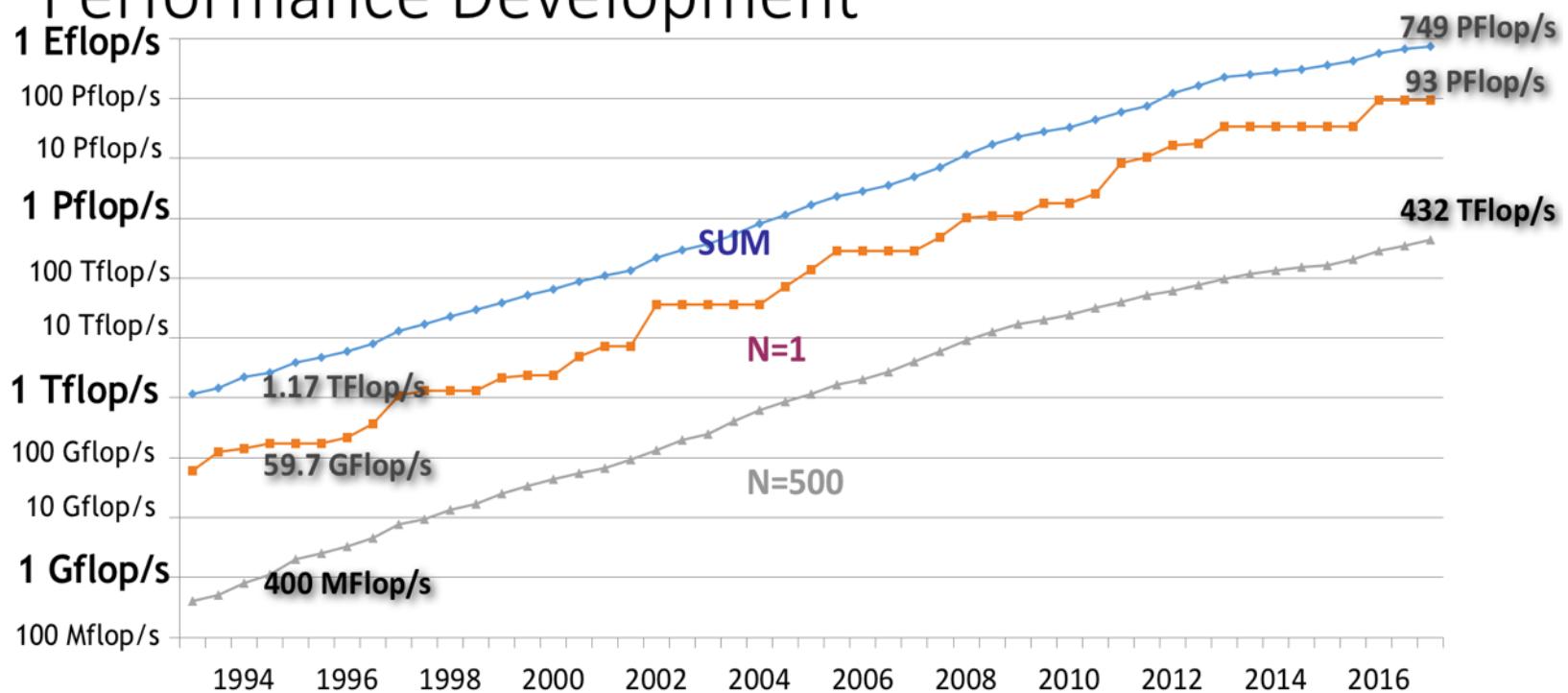


Titan  
17.59 Pflops



Taihulight  
93 Pflops

# Performance Development



The evolution of the  $R_{\max}$  from the HPL benchmark for supercomputing systems in the Top 500 list since the beginning of the list in 1993. Blue (top line) indicates the cumulative performance of all the computers in the list. Red (middle line) shows the performance of the #1 computer in the list. Orange (bottom line) shows the performance of the #500 computer in the list.

# 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

# HPC Modalities

- Capacity Computing
  - Also referred to as “throughput computing”
  - Job stream parallelism: a form of weak scaling
  - No interrelationship between independent tasks
    - No inter-synchronization, No data exchange
  - Performance measured by total floats across jobs normalized to time
- Capability Computing
  - Single job, fixed-size data set: strong scaling
  - Tightly coupled, strong interrelationship among parallel actions
  - Performance achieved through reduction of time to solution
- “Cooperative Computing”
  - Single job, weak scaling: problem data set expands to fill scaled up systems
  - Coordinated via shared synchronization
  - Cooperative through exchange of intermediate result values



# HPC Examples



Tianhe-2

55 Petaflops peak performance  
33.9 Petaflops Linpack Rmax  
1,375 Terabytes memory  
Intel Xeon Phi Accelerator  
24 Mwatts power  
NUDT deployed  
Inspur manufacturer

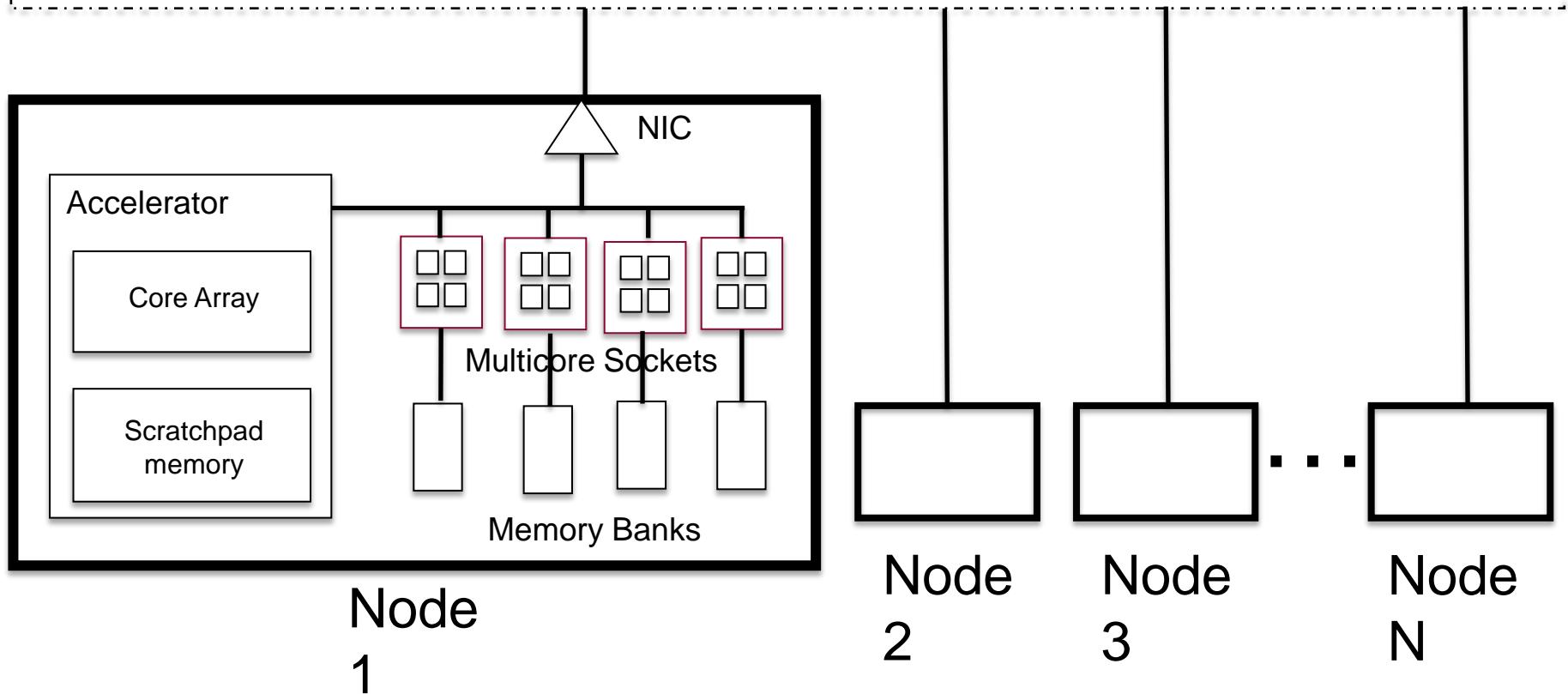


Titan

27 Petaflops peak performance  
17.5 Petaflops Linpack Rmax  
693 Terabytes memory  
NVIDIA Tesla Accelerator GPU  
8.2 MWatts power  
ORNL deployed  
Cray manufacturer

# Conventional Heterogeneous Multicore System Architecture

## Global Interconnection Network



# Machine Parameters Affecting Performance

- Peak floating point performance
- Main memory capacity
- Bi-section bandwidth
- I/O bandwidth
- Secondary storage capacity
- Organization
  - Class of system
  - # nodes
  - # processors per node
  - Accelerators
  - Network topology
- Control strategy
  - MIMD
  - Vector, PVP
  - SIMD
  - SPMD

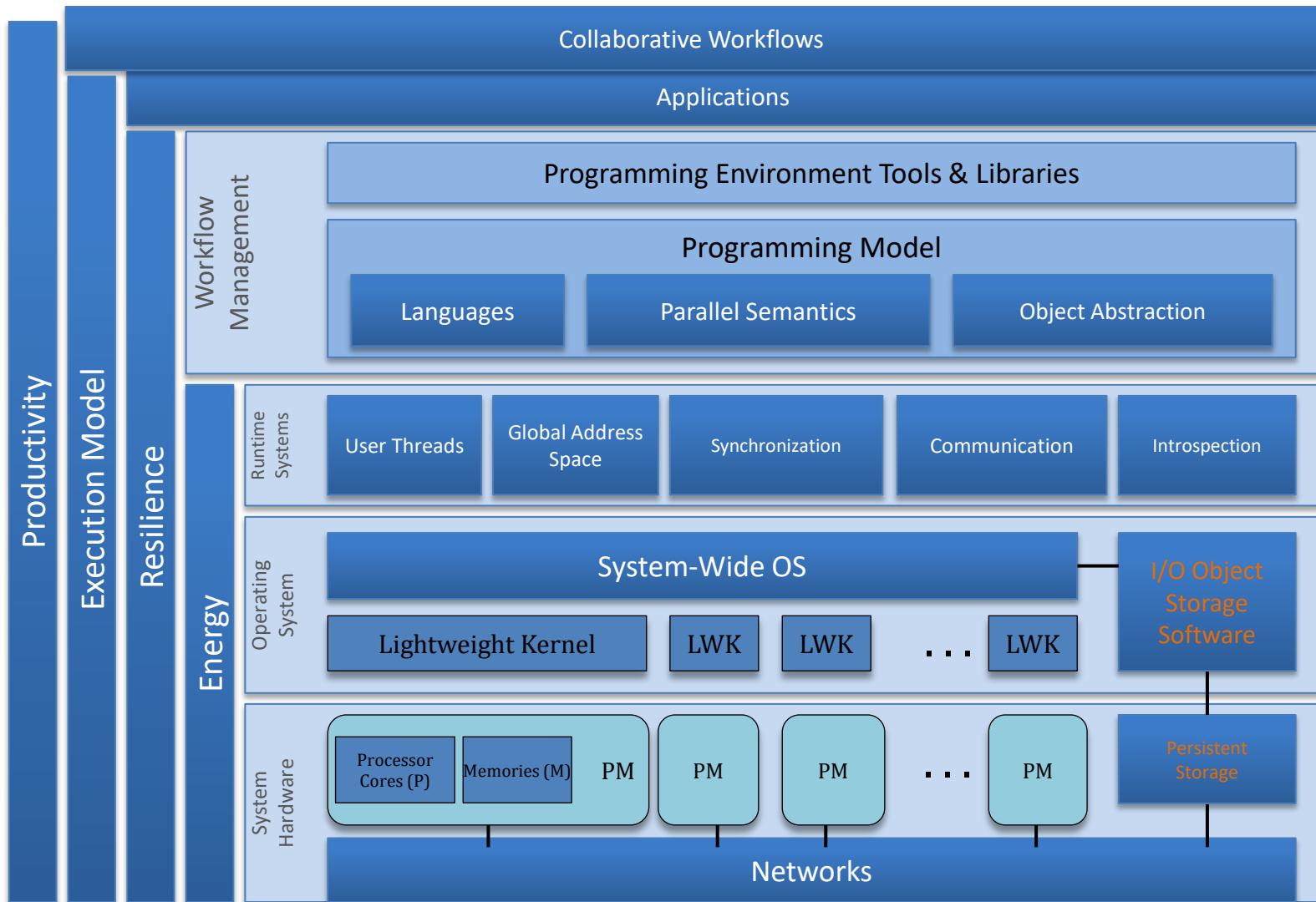


# Titan Specifications

- Performance
  - Peak: 27 Petaflops
  - HPL: 17.59 Petaflops
- Storage Capacity
  - Memory: 710 Terabytes
  - Secondary: 10 Petabytes
- Physical
  - # racks: 200 cabinets
  - Floor space: 4,452 sq. feet
  - Power: 8.2 Megawatts
- Cost: \$97M Architecture
  - Cray XK7
  - CPU: AMD Opteron 6274
    - 16 cores
  - GPU: Nvidia Tesla K20X
    - 2,496 Cuda cores
    - 732 MHz clock
  - Node: 1 CPU + 1 GPU
    - 18,688 total
    - 2 nodes per blade

# Supercomputing System Stack

- Applications
  - End user problems, often in sciences and technology
- Algorithms
  - Numerical techniques
  - Means of exposing parallelism
- Programming
  - Languages, tools, & environments
- Compilers and runtime software
  - Maps application program to system resources, mechanisms, and semantics
- Operating systems
  - Manages resources and provides virtual machine
- Computer architecture
  - Semantics and structures
- Device technologies
  - Enabling technologies for logic, memory, & communication
  - Circuit design



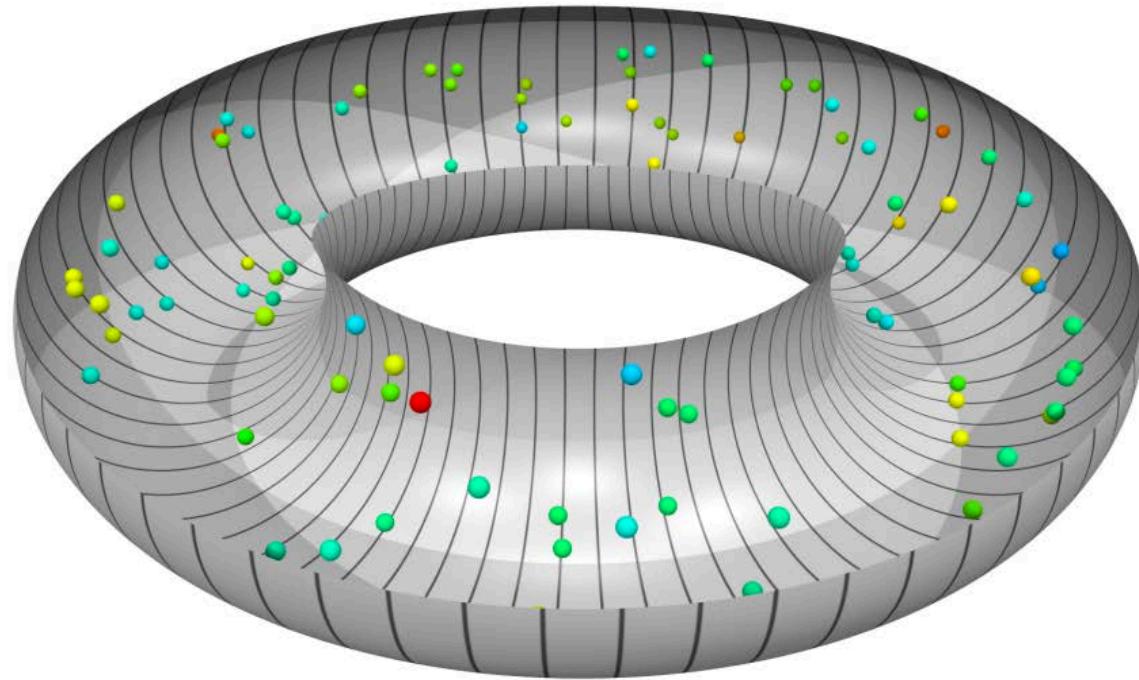
The system stack of a general supercomputer consists of a system hardware layer and several software layers. The first of the software layers is the operating system encompassing both resource management and middleware to access I/O channels. Higher software layers include runtime systems and workflow management.

# Programming Models and Interfaces



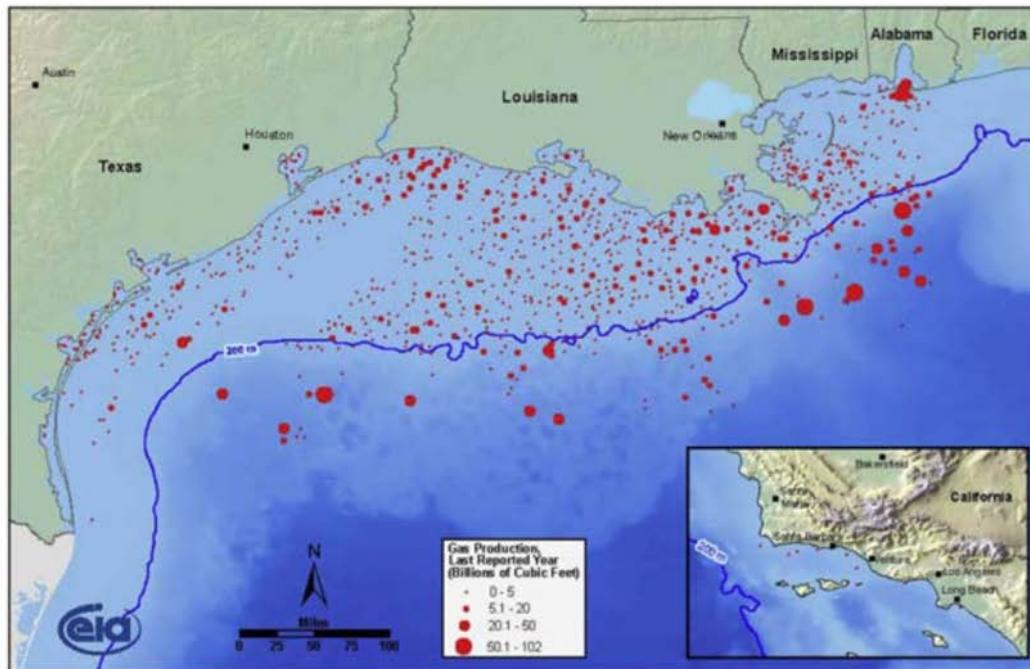
- Sequential
  - Fortran
  - C
  - C++
- Capacity/Throughput
  - Job stream parallelism
  - Condor
  - Moab
- Cooperative
  - Communicating Sequential Processing
  - MPI
- Capability
  - Shared memory, multiple threads
  - OpenMP
  - Cilk++

Supercomputing Problem Representatives	Academia	Industry	Government
<b>Solution of Partial Differential Equations</b>	Navier-Stokes equations, Einstein equations, Maxwell Equations	Black-Scholes Equation, Navier-Stokes equations for compressible flow, oil reservoir modeling	Weather prediction, hurricane modeling, storm surge modeling, sea ice modeling
<b>Large systems with pairwise force interactions</b>	Cosmology, molecular dynamics simulations	Medicine development, biomolecular dynamics	Plasma modeling
<b>Linear Algebra</b>	Supporting the solution of partial differential equations, fundamental benchmarks HPL and HPCG	Search engine PageRank, finite element simulations	HPC machine evaluation, climate modeling
<b>Graph Problems</b>	Systems research, machine learning	Fraud detection	Security services, data analytics
<b>Stochastic Systems</b>	Radiation transport, particle physics	Risk analysis in finance, nuclear reactor design, process control	Public health, modeling spread of disease

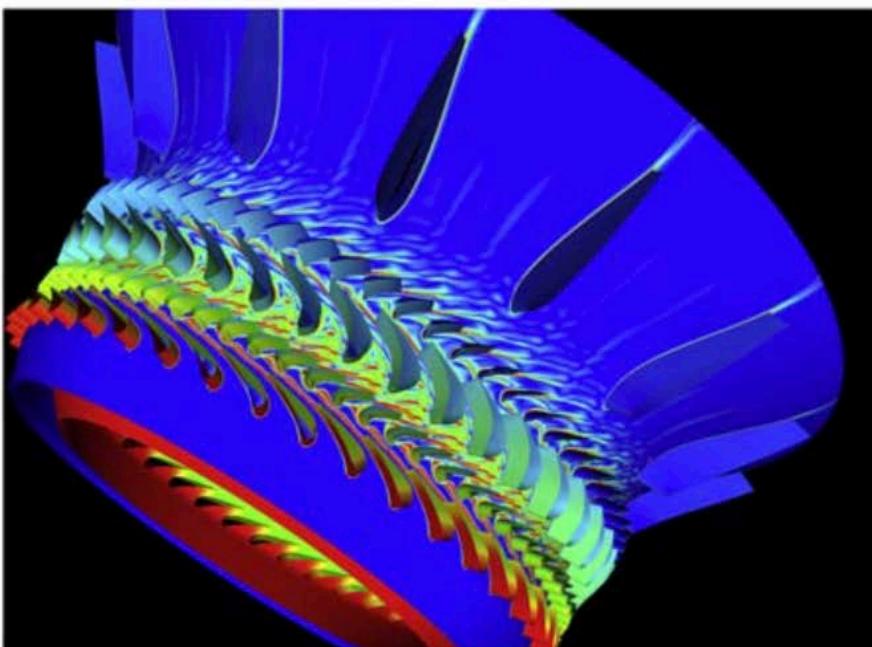
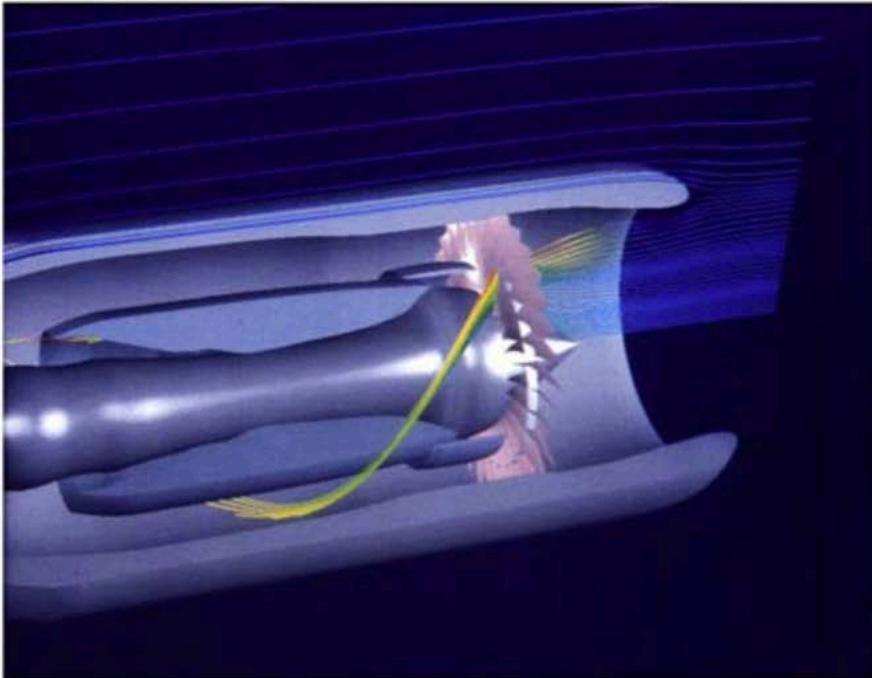


A Particle-In-Cell simulation from the Gyrokinetic Toroidal code (Princeton Plasma Physics laboratory) that simulates a plasma within a Tokomak fusion device. A sampling of some particles within the toroid are shown here colored according to their velocity with different supercomputing processor boundaries delineated by the toroidal subdivisions.

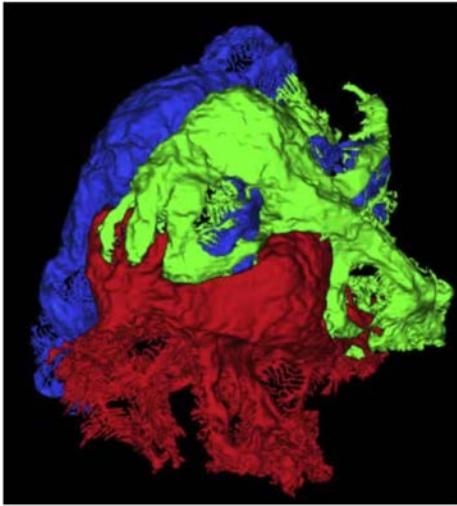
## Gas Production in Offshore Fields, Lower 48 States



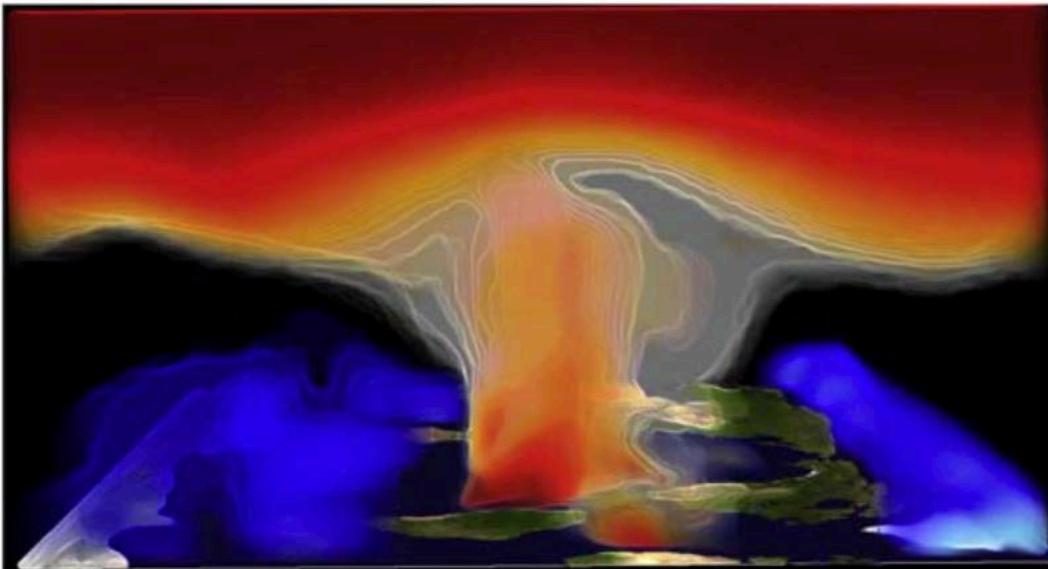
Researchers at BP use HPC to simulate sub-surface geologies using multi-dimensional analysis and characterization to accurately identify oil reservoirs.



HPC is frequently used in high-fidelity virtual engine simulation and design. (Bottom) Researchers at NASA use HPC to simulate the design of next-generation turbines for both aviation and power production.



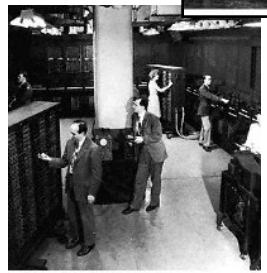
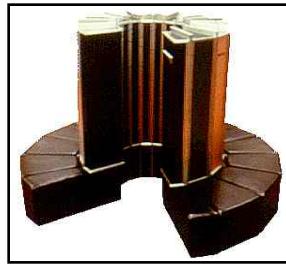
HPC is used to develop virtual models of kidney podocytes.



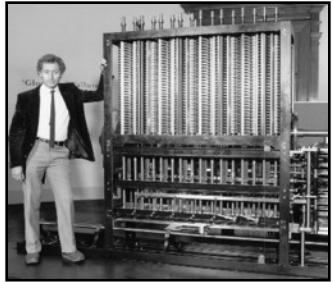
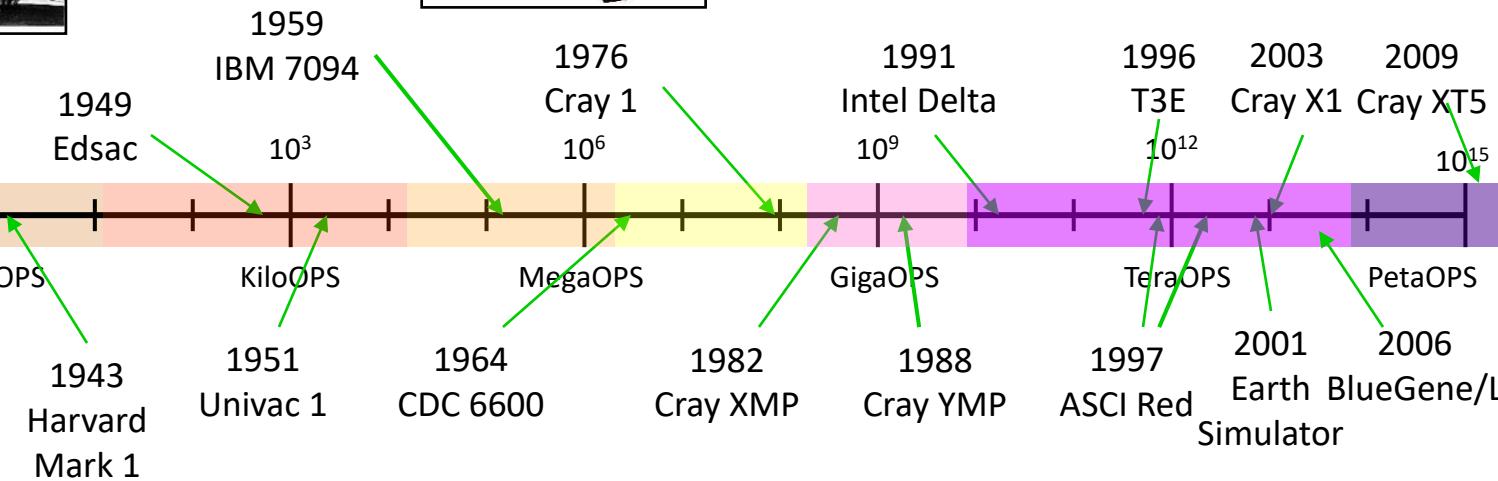
Researchers at Oak Ridge National Laboratory explore the advection of carbon dioxide in an atmospheric model.

<u>Vertical Segments</u>	<u>Common Workflows</u>
<u>Financial Services</u>	<u>Fraud and Anomaly Detection, Backtesting for Algorithmic / proprietary trading, Risk Analytics</u>
<u>Oil and Gas</u>	<u>Seismic Processing, Interpretation , Reservoir modeling</u>
<u>Manufacturing</u>	<u>Materials Simulation, Structural Simulations (Noise/Vibration/Hardness &amp; Crash), Aerodynamics simulations, Design space exploration, Thermal simulations and many more</u>
<u>Life Sciences</u>	<u>Molecular Dynamics, Drug discovery, Virtual modeling, genome sequencing and many more.</u>
<u>Earth Sciences</u>	<u>Atmospheric modeling, Hydrodynamic modeling, Ice Modeling, Coupled Climate Modeling</u>

# History of HPC



1823  
Babbage Difference  
Engine

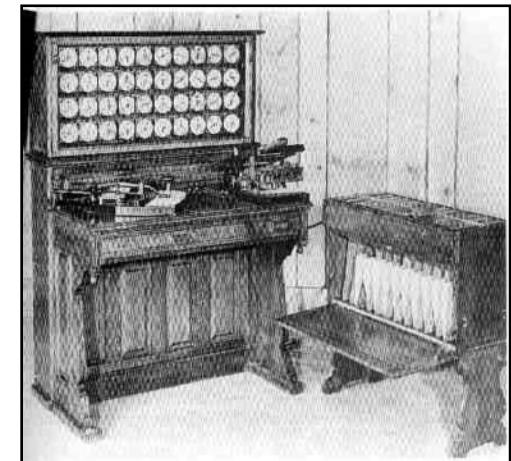


# Epoch I: Mechanical Calculating

- 1650 – 1950
- Enabling technology
  - Tally sticks, beads
  - Gears, carry mechanism
  - Punched cards
- Fundamental Concepts
  - Storage of numeric values
  - Method of arithmetic ops
  - Instruction sequence control
- Accomplishments
  - Abacus, Pascaline, Jacquard Loom
  - Design: Babbage Analytical Engine
  - 1890 U.S. census
  - Tabulator, Harvard Mark 1, ENIAC, Colossus
  - 1 ops performance



Pascal's calculator



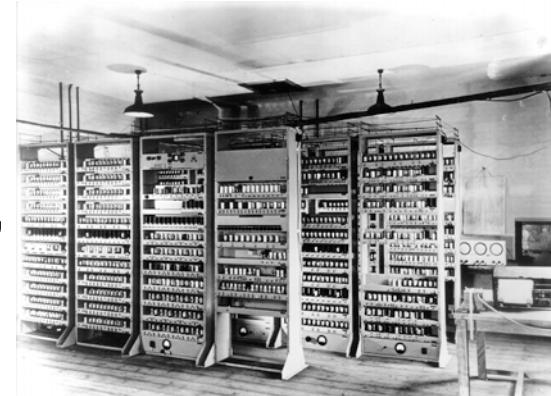
Hollerith's punched card tabulator

# Epoch II: von Neumann Architecture

- 1950 – 1960
- Enabling Technology
  - Vacuum Tube
    - Signal amplification
    - Logic circuits
    - Flip-flop bit latch
  - Memory
    - Mercury delay line
    - Electro-static Williams tube
    - Magnetic cores



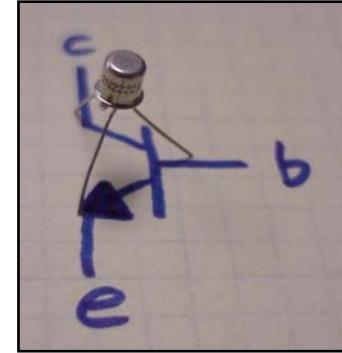
EDSAC,  
1949



- Fundamental Concepts
  - Von Neumann execution model
  - Turing computability
  - Boolean logic
  - Bit identified by Shannon
- Accomplishments
  - KIPS performance
  - EDSAC, UNIVAC, Whirlwind, IBM 709

# Epoch III: Instruction Parallelism

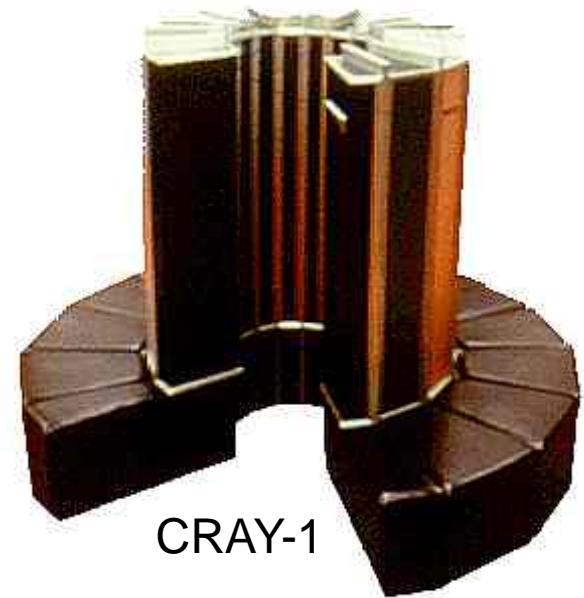
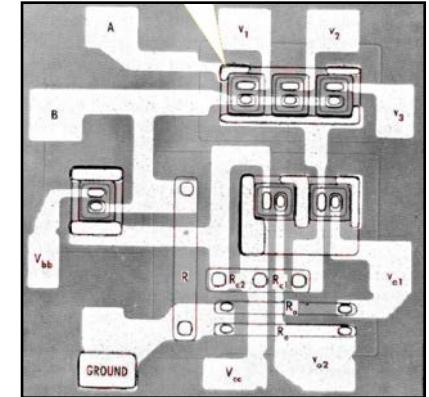
- Enabling technology
  - Transistor
  - Semiconductor: Germanium, Silicon
- Fundamental Concepts
  - Improve clock rate through technology
  - Parallelism: Overlapped instruction execution
- Accomplishments
  - Replace vacuum tube with transistor
  - Architecture parallelism exploited
  - CDC 6600 delivered 1964 to Los Alamos
  - 1 Megaflops peak performance



CDC 6600

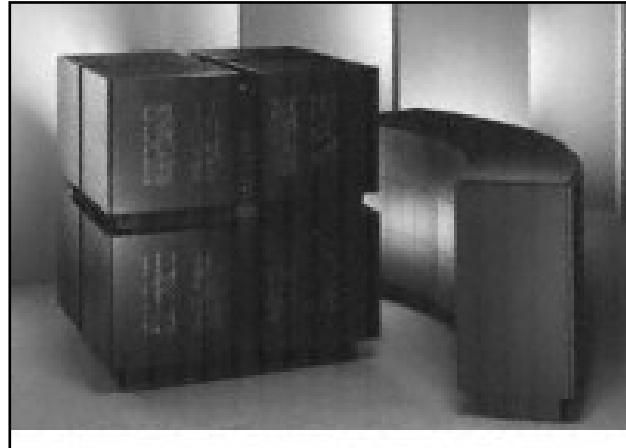
# Epoch IV: Vector Processing

- 1975 – 1980
- Enabling Technology
  - SSI (Small Scale Integration), MSI (Medium Scale Integration)
  - Integrated Circuits: RTL, DTL, TTL, ECL
- Fundamental Concepts
  - Increased clock rate
    - Technology
    - Logic design – small functional steps
  - Pipeline structures
  - Vector execution model
- Accomplishments
  - Cray 1
  - 80 MHz clock rate
  - 136 Megaflops Peak Performance



# Epoch V: SIMD-array/PVP

- 1980 – 1990
- Enabling technology
  - LSI (Large Scale Integration)
- Fundamental Concepts
  - Single Instruction Stream,  
Multiple Data
  - Medium grained parallelism
  - Parallel Vector Processing (PVP)
  - Array of numeric processors  
under single control stream
- Accomplishments
  - Gigaflops range performance
  - Maspar, Cray XMP, YMP, ...



# Epoch VI: CSP

- 1990 - 2005
- Enabling Technology
  - VLSI (Very Large Scale Integration)
  - 32-bit/64-bit Microprocessor (the Killer Micro)
  - System Area Network (SAN)
- Fundamental Concepts
  - Communicating Sequential Processing (CSP)
  - Distributed memory, message passing
  - Bulk Synchronous Parallel (BSP)
- Accomplishments
  - Commodity clusters and MPPs
  - Dominates for two decades
  - From 10s of Gigaflops to Teraflops to Petaflops performance



Intel Touchstone Delta



Beowulf Commodity Cluster

# Epoch VII: Multicore/Heterogeneous

- 2005 - 2020
- Enabling Technology
  - Multi-core
- Fundamental Concepts
  - Hybrid execution
  - Heavy-weight nodes
- Accomplishments
  - Fastest computers in the world
  - Improved energy efficiency
  - Hybrid programming interfaces



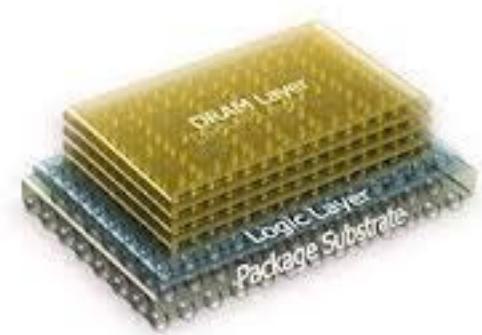
Cray/ORNL Titan



NUDT Tianhe-2

# Epoch VIII: Exascale

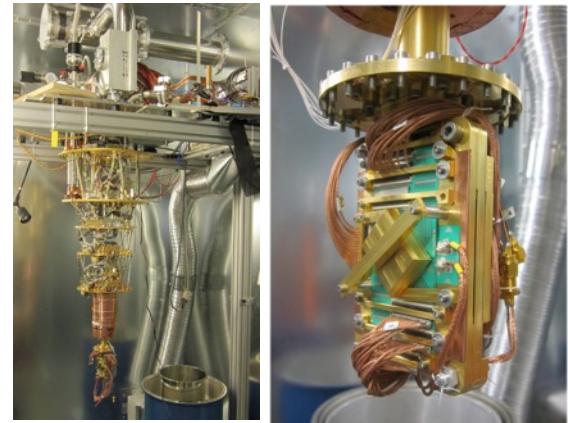
- 2020 - 2030
- Enabling Technology
  - 3-D die stacking
  - Optical inter/intra socket networking
  - End of Moore's Law
  - Processor-in-Memory (PIM)
- Fundamental Concepts
  - Innovative execution models
  - Dynamic adaptive resource management
  - Message-driven computation
  - Multi-threading
- Accomplishments
  - Billion-way parallelism
  - Energy <20pJ/op
  - Runtime system software



3-D Die Stacking

# Epoch IX: Quantum ?

- Beyond 2030
- Enabling Technology
  - Cryogenics
  - Josephson Junctions
  - Graphene and NCT
- Fundamental Concepts
  - Quantum Mechanics
- Accomplishments
  - A few qubits
  - Adiabatic switching
  - Algorithms
  - Witnessing entanglement



Qubits



D-Wave Quantum  
Computer

# Parallel Computer Architecture

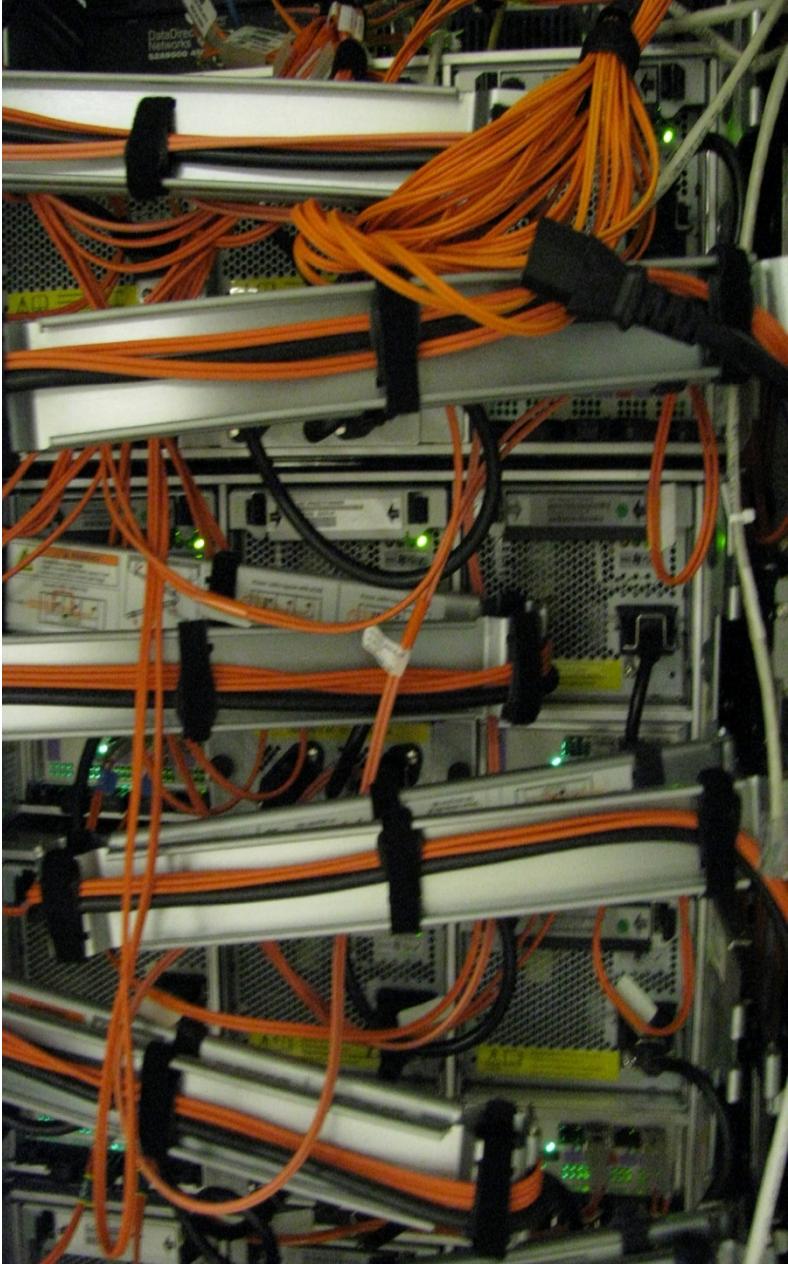


# Opening Remarks

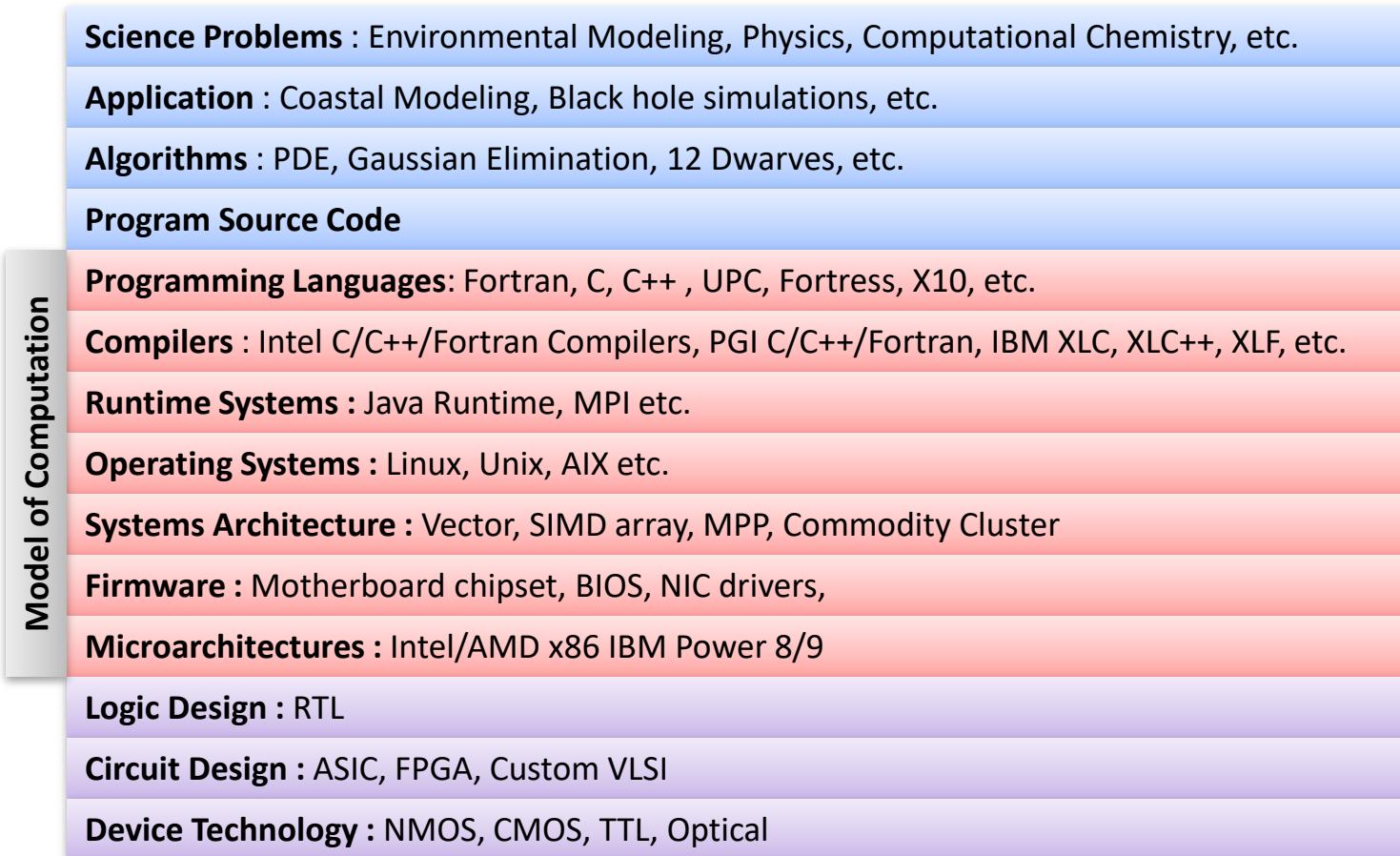
- This lecture is an introduction to supercomputer architecture
  - Major parameters, classes, and system level
- Architecture exploits device technology to deliver its innate computation performance potential
  - Structures and system organization
  - Semantics of operation and memory (instruction set architecture, ISA)
- Between device technology and architecture is circuit design
  - Circuit design converts devices to logic gates and higher level logical structures (e.g. multiplexers, adders)
  - but this is outside the scope of this course.
- We will assume basic logic abstraction with characterizing properties:
  - Functional behavior (the logical operation it performs)
  - Switching speed
  - Propagation delay or latency
  - Size and power

# Topics

- The HPC System Stack
- What is Computer Architecture
- Review Performance Factors and Metrics
- MIMD class HPC Architecture
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architecture

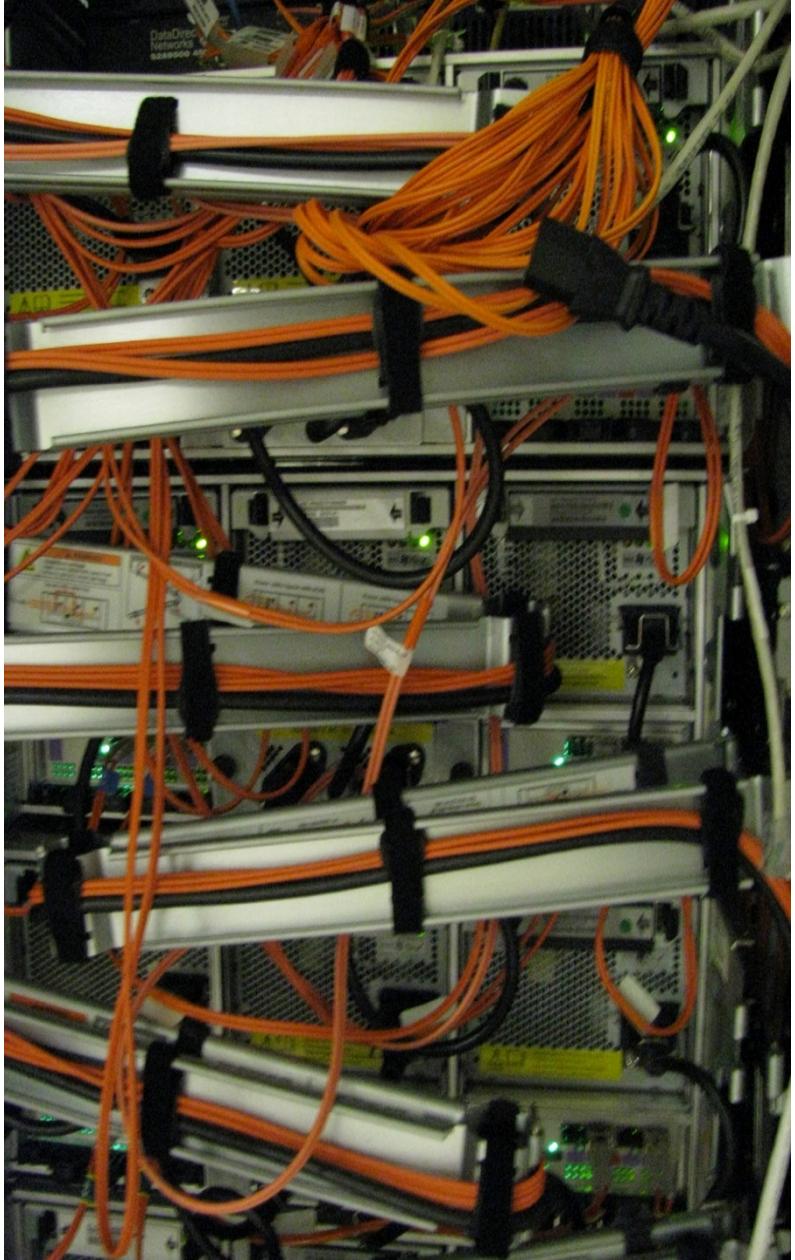


# HPC System Stack



# Topics

- The HPC System Stack
- What is Computer Architecture
- Review Performance Factors and Metrics
- MIMD class HPC Architecture
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architectures



# Computer Architecture

- Structure
  - Functional elements
  - Organization and balance
  - Interconnect and Data flow paths
- Semantics
  - Meaning of the logical constructs
  - Primitive data types
  - Manifest as Instruction Set Architecture abstract layer
- Mechanisms
  - Primitive functions that are usually implemented in hardware or sometimes firmware
  - Determines preferred actions and sequences
  - Enables efficiency and scalability
- Policy
  - Approach and priorities to accomplishing a goal
  - e.g., cache replacement policy

# Structure

- Functional elements
  - The form of functional elements made up of more primitive logical modules
  - e.g. vector arithmetic unit comprising a pipeline of simple stages
- Organization and balance
  - Number of major elements of different types
  - Hierarchy of collections of elements
- Data flow
  - Interconnection of functional, state, and communication elements
  - Control of dataflow paths determines actions of processor and system

# Semantics

- Meaning of the logical constructs
  - Basic operations that can be performed on data
- Primitive data types
  - What collections of bits (e.g. word) means
  - Defines actions that can be performed on binary strings
- Instruction Set Architecture
  - Defined set of actions that can be performed and data object on which they can be applied
  - Encoding of binary strings to represent distinct instructions
- Parallel control constructs
  - Hardware implemented : vector operations,
  - Software implemented : MPI libraries

# Mechanisms

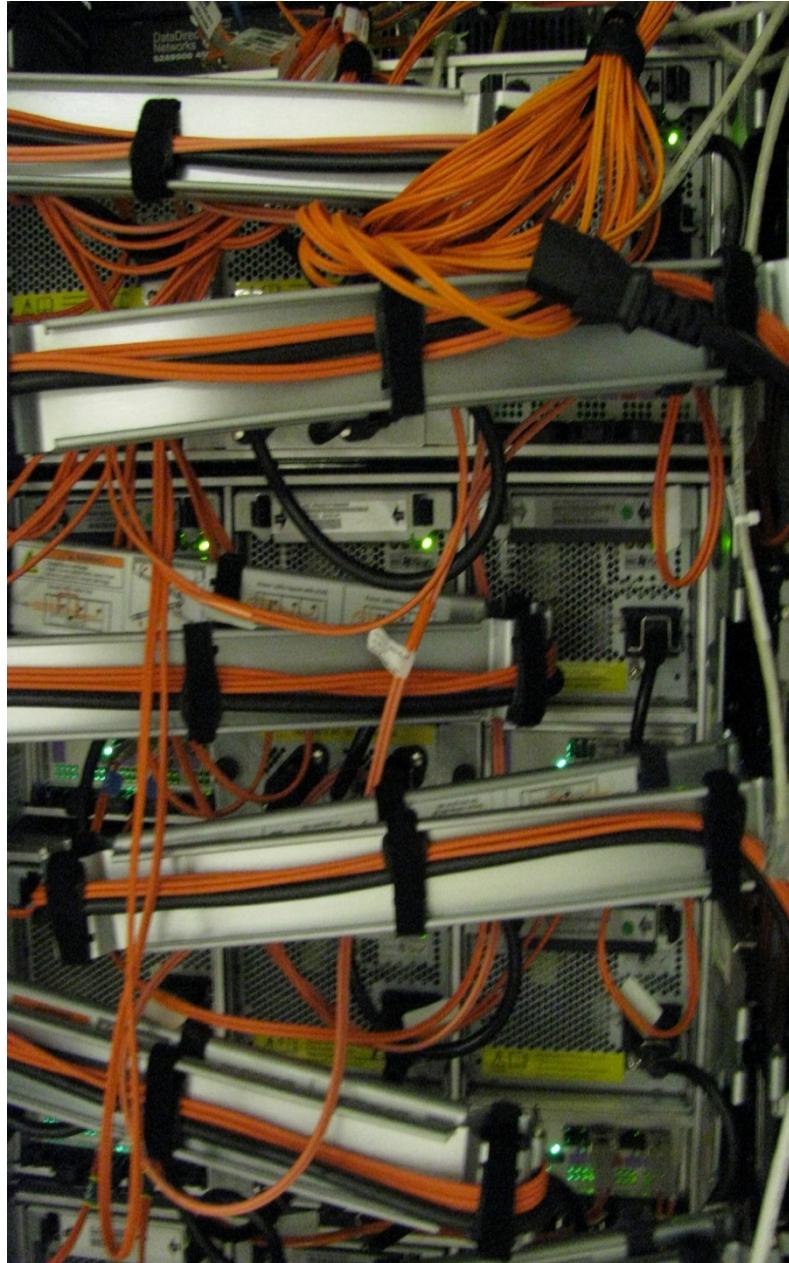
- Primitive functions that are usually implemented in hardware or sometimes firmware
  - Lower level than instruction set operations
  - Multiple such mechanisms contribute to execution of operation
- Determines preferred actions and sequences
  - Usually time effective primitives
  - Usually widely used by many instructions
- Enables efficiency and scalability
  - Establishes basic performance properties of machine
- Examples
  - Basic arithmetic and logic unit functions
  - Thread context switching
  - TLB (Translation Lookaside Buffer) address translation
  - Cache line replacement
  - Branch prediction

# Policy

- Hardware architecture policies
  - Decision of ordering or allocation dependent on criteria
  - Not all machine decisions are visible to the ISA of the system
  - Not all machine choices are available to the name space of the operands
  - Examples
    - Cache structure, size, and speed
    - Cache replacement policies
    - Order of operation execution
    - Branch prediction
    - Allocation of shared resources
    - Network routers
- Software system management policies
  - Scheduling,
  - Data allocation : partitioning of a problem

# Topics

- The HPC System Stack
- What is Computer Architecture
- **Review Performance Factors and Metrics**
- MIMD class HPC Architecture
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architectures



# Performance Factors: Technology Speed

- Latencies
  - Logic latency time
  - Processor to memory access latency
  - Memory access time
  - Network latency
- Cycle Times
  - Logic switching speed
  - On-chip clock speed (clock cycle time)
  - Memory cycle time
- Throughput
  - On-chip data transfer rate
  - Instructions per cycle
  - Network data rate
- Granularity
  - Logic density
  - Memory density
  - Task Size
  - Packet Size

# Machine Parameters affecting Performance

- Peak floating point performance
- Main memory capacity
- Bi-section bandwidth
- I/O bandwidth
- Secondary storage capacity
- Organization
  - Class of system
  - # nodes
  - # processors per node
  - Accelerators
  - Network topology
- Control strategy
  - MIMD
  - Vector, PVP
  - SIMD
  - SPMD

# Performance Factors: Parallelism

- Fully independent processing elements operating concurrently on separate tasks
  - Coarse grained
  - Communicating Sequential Processes (CSP),
  - Single Program Multiple Data stream (SPMD)
- Instruction Level Parallelism (ILP)
  - Fine grained
  - Single instruction performs multiple operations
- Pipelining
  - Fine grained
  - Overlapping sequential operations in execution pipeline
  - Vector pipelines
- SIMD operations
  - Fine / Medium grained
  - Single Instruction stream, **M**ultiple Data stream
  - ALU arrays
- Overlapping of computation and communication
  - Fine / Medium grained
  - Asynchronous
  - Prefetching
- Multithreading
  - Medium grained
  - Separate instruction streams serve single processor

# Sources of Performance Degradation (SLOW)

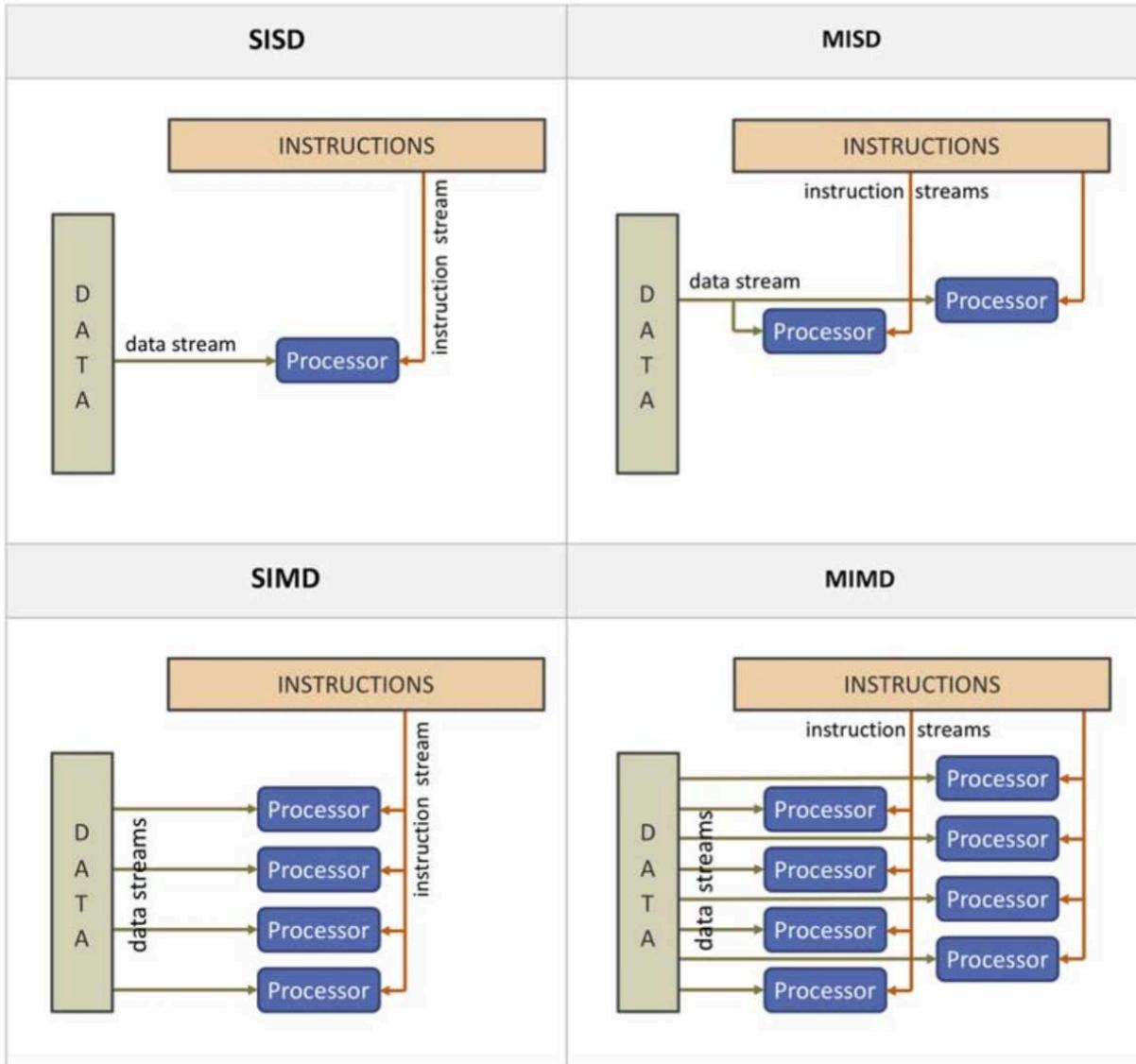
- Starvation
  - Not enough work to do among distributed resources
  - Insufficient parallelism
  - Inadequate load balancing
  - e.g. : Amdahl's law
- Latency
  - Time required for response of access to remote data or services
  - Waiting for access to memory or other parts of the system
  - e.g. : Local memory access, Network communication
- Overhead
  - Extra work that has to be done to manage program concurrency and parallel resources the real work you want to perform
  - Critical-path work for management of concurrent tasks and parallel resources not required for sequential execution
  - e.g. : Synchronization and scheduling
- Waiting for Contention
  - Delays due to conflicts for use of shared resources.
  - e.g. : Memory bank conflicts, shared network channels

# Parallel Structures & Performance Issues

- Pipelining
  - Vector processing
  - Execution pipeline
  - Performance Issues:
    - Pipelining increases throughput : More operations per unit time
    - Pipelining increases latency time : Operation on single operand pair can take longer than non-pipelined functional unit
- Multiple Arithmetic Units
  - Instruction level parallelism
  - Systolic arrays
  - Performance Issues:
    - Increases peak performance
    - Requires application instruction level parallelism
    - Average usually significantly lower than peak

# Parallel Structures & Performance Issues

- Multiple processors
  - MIMD: Separate control
  - SIMD: Single controller
  - Multicore
  - Accelerators
- Performance Issues: Multiple processors require overhead operations
  - Synchronization
  - Communications
  - Possibly cache Coherence



Flynn's taxonomy. MIMD, multiple instruction stream, multiple data stream; MISD, multiple instruction stream, single data stream; SIMD, single instruction stream, multiple data stream; SISD, single instruction stream, single data stream.

# Scalability

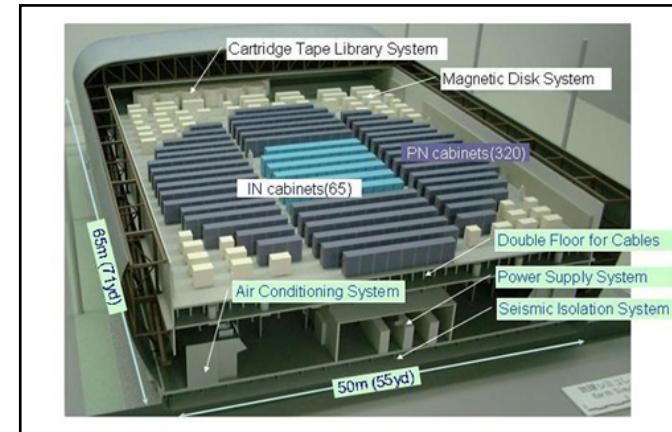
- The ability to deliver proportionally greater sustained performance through increased system resources
- Strong Scaling
  - Fixed size application problem
  - Application size remains constant with increase in system size
- Weak Scaling
  - Variable size application problem
  - Application size scales proportionally with system size
- Capability computing
  - in most pure form: strong scaling
  - Marketing claims tend toward this class
- Capacity computing
  - Throughput computing
  - Includes job-stream workloads
  - In most simple form: weak scaling
- Cooperative computing
  - Interacting and coordinating concurrent processes
  - Not a widely used term
  - Also: “coordinated computing”

# Performance Metrics

- Peak floating point operations per second (flops)
- Peak instructions per second (ips)
- Sustained throughput
  - Average performance over a period of time
  - flops, Mflops, Gflops, Tflops, Pflops
  - flops, Megaflops, Gigaflops, Teraflops, Petaflops
  - ips, Mips, ops, Mops ...
- Cycles per instruction
  - cpi
  - Alternatively: instructions per cycle, ipc
- Memory access latency
  - cycles per second
- Memory access bandwidth
  - bytes per second (Bps)
  - bits per second (bps)
  - or Gigabytes per second, GBps, GB/s
- Bi-section bandwidth
  - bytes per second

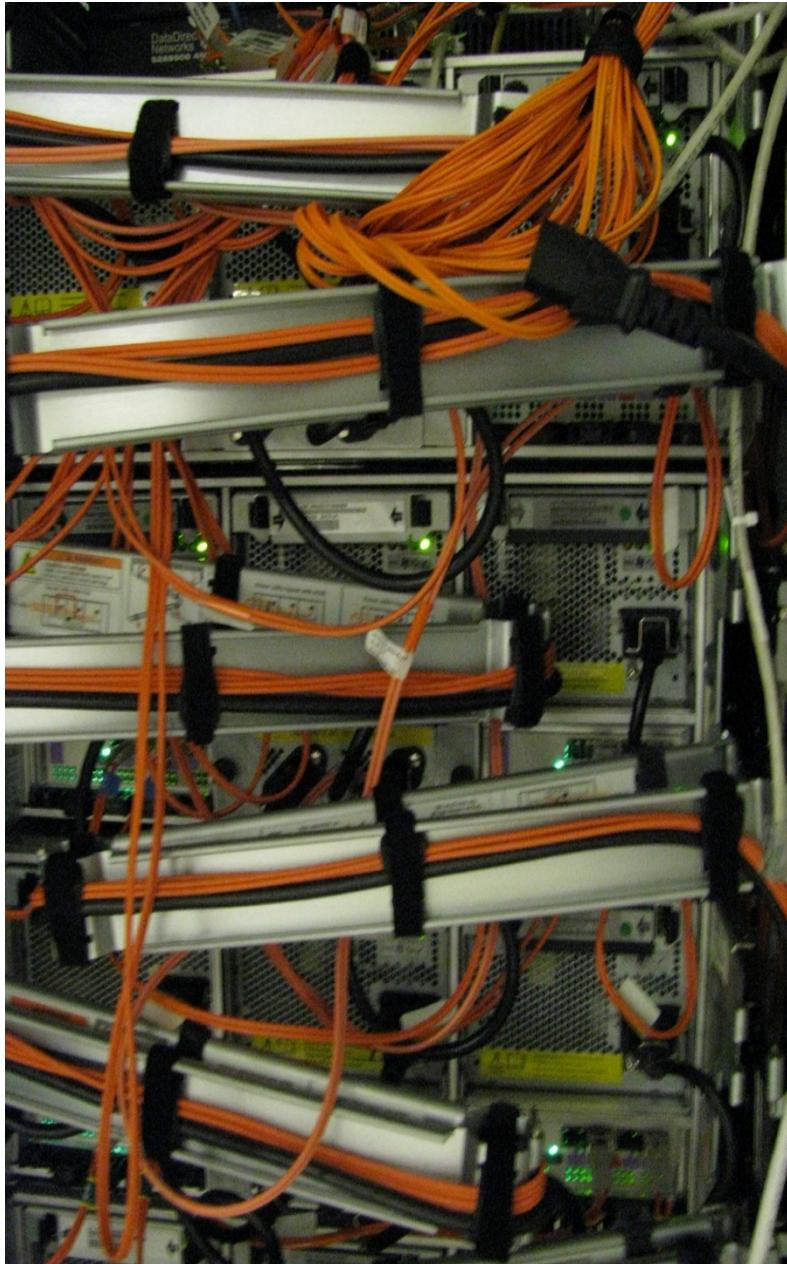
# Practical Constraints and Limitations

- Cost
  - Deployment
  - Operational support
- Power
  - Energy required to run the computer
  - Energy for support facilities
  - Energy for cooling (remove heat from machine)
- Size
  - Floor space
  - Access way for power and signal cabling
- Reliability
  - One factor of availability
- Generality
  - How good is it across a range of problems
- Usability
  - How hard is it to program and



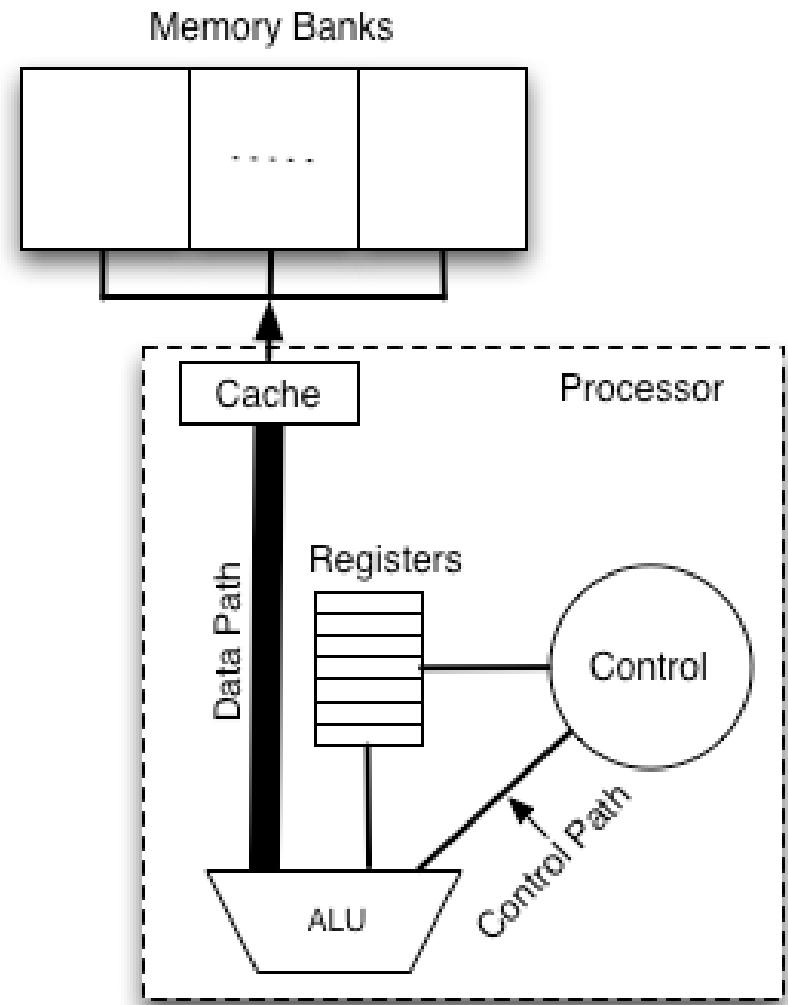
# Topics

- The HPC System Stack
- What is Computer Architecture
- Review Performance Factors and Metrics
- **MIMD class HPC Architecture**
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architectures



# Basic Uni-processor Architecture elements

- I/O Interface
- Memory Interface
- Cache hierarchy
- Register Sets
- Control
- Execution pipeline
- Arithmetic Logic Units



# Multiprocessor

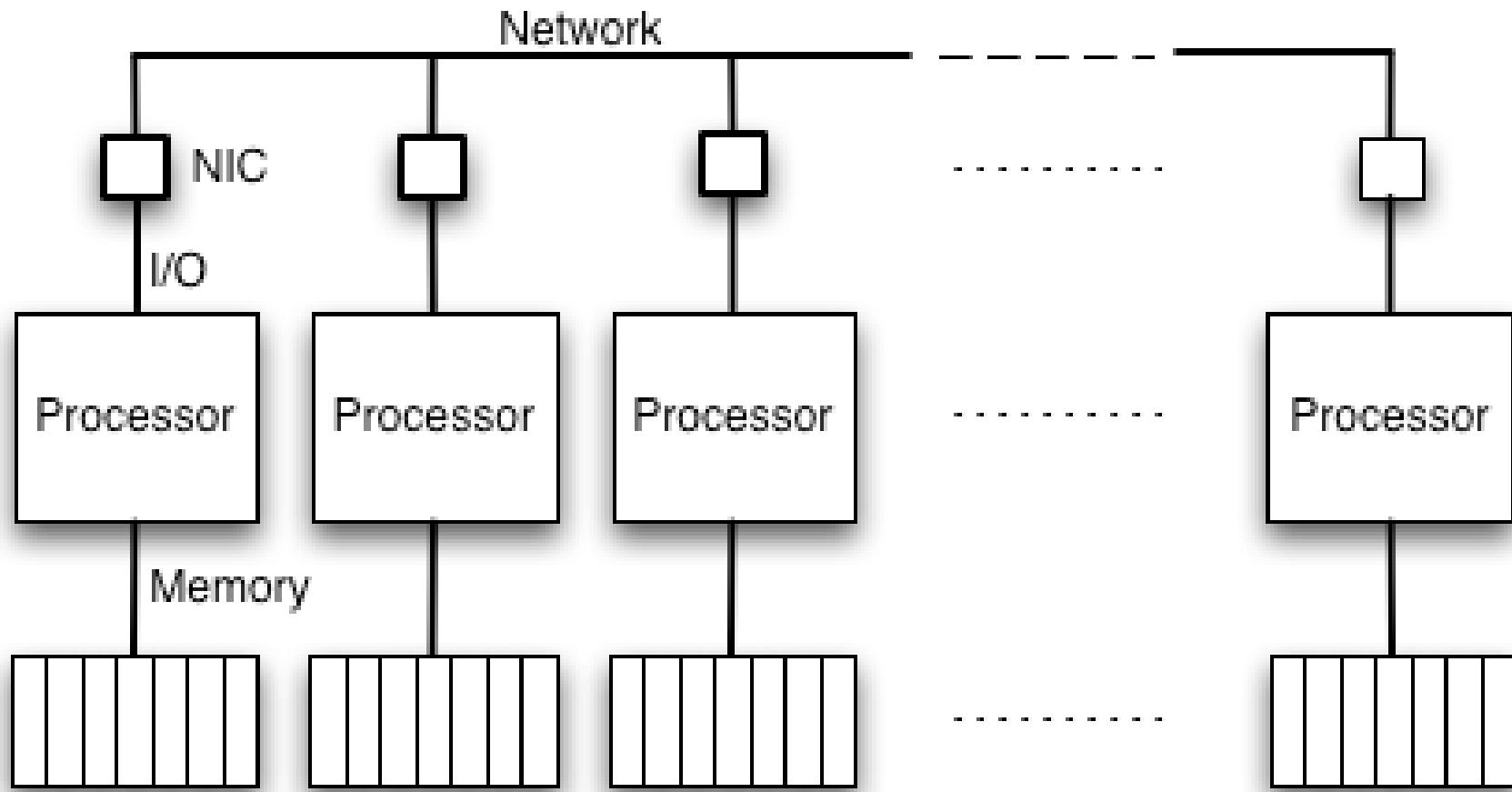
- A general class of system
- Integrates multiple processors in to an interconnected ensemble
- MIMD: Multiple Instruction Stream Multiple Data Stream
- Different memory models
  - Distributed memory
    - Nodes support separate address spaces
  - Shared memory
    - Symmetric multiprocessor
    - UMA – uniform memory access
    - Cache coherent
  - Distributed shared memory
    - NUMA – non uniform memory access
    - Cache coherent
  - PGAS
    - Partitioned global address space
    - NUMA
    - Not cache coherence
  - Hybrid : Ensemble of distributed shared memory nodes
    - Massively Parallel Processor, MPP

# Massively Parallel Processor

- MPP
- General class of large scale multiprocessor
- Represents largest systems
  - IBM BG/L
  - Cray XT3
- Distinguished by memory strategy
  - Distributed memory
  - Distributed shared memory
    - Cache coherent
    - Partitioned global address space
- Custom interconnect network
- Potentially heterogeneous
  - May incorporate accelerator to boost peak performance

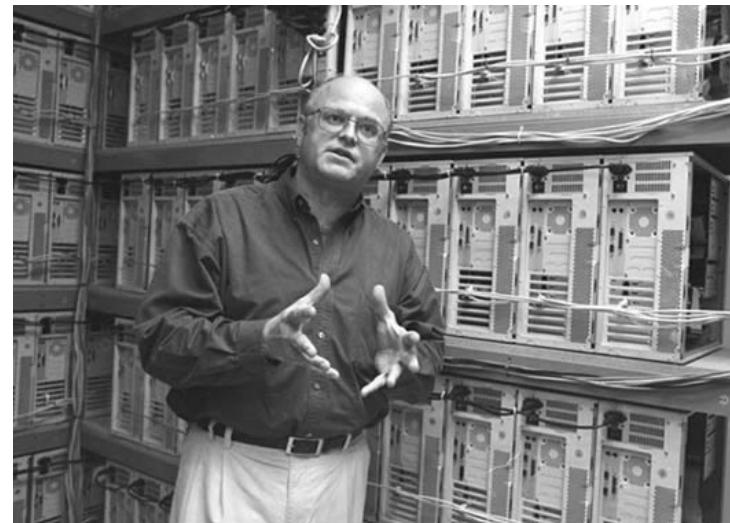
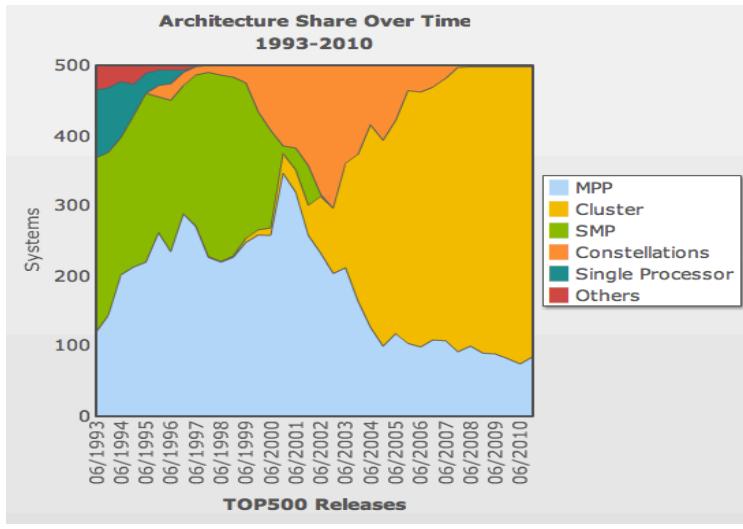


# DM-MPP



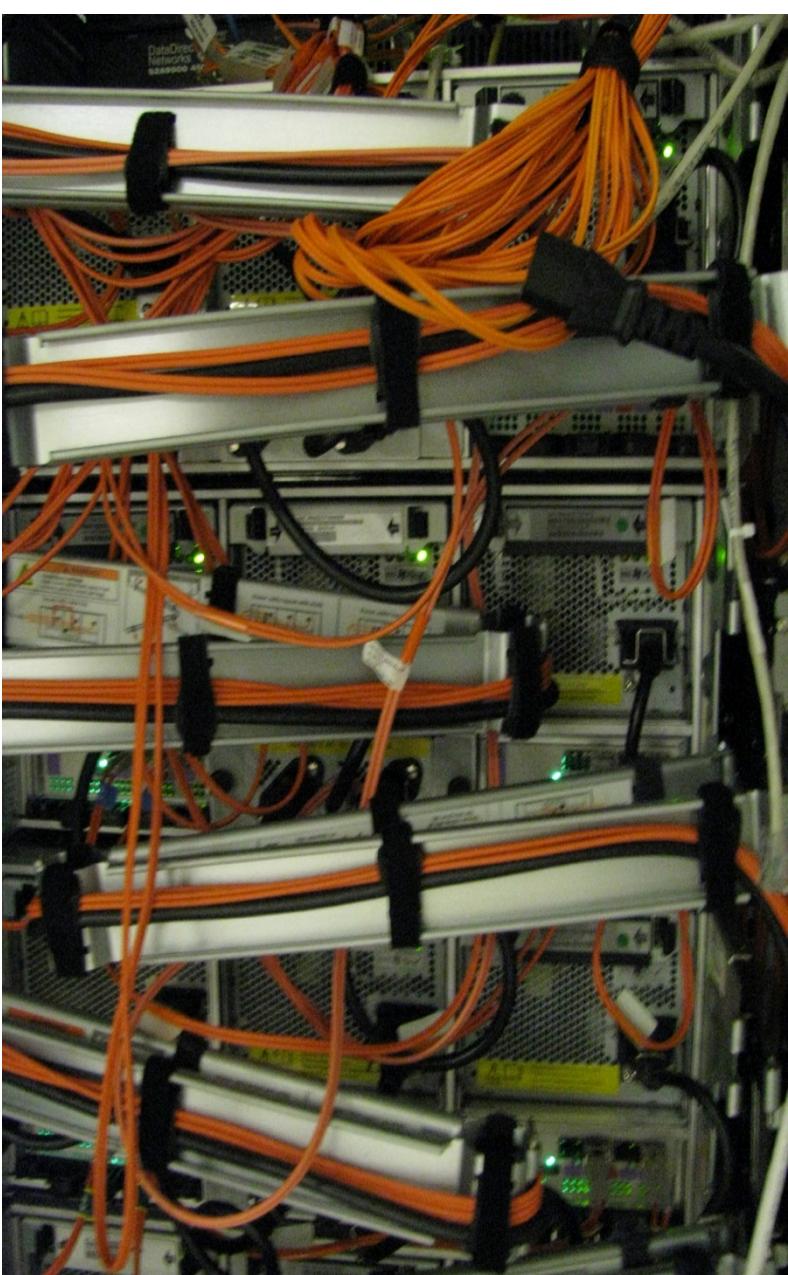
# Commodity Clusters

- Distributed Memory systems
- Superior performance to cost
- Dominant parallel systems architecture on the Top 500 List
- Combines off the shelf systems in scalable structure
- Employs commercial high-bandwidth networks for integration
- Message Passing programming model used (e.g. MPI)
- First cluster on Top500 : Berkley Now, 1997



# Topics

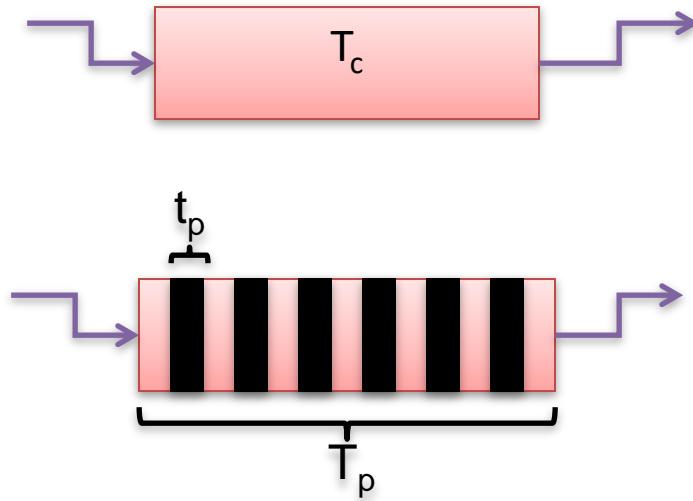
- The HPC System Stack
- What is Computer Architecture
- Review Performance Factors and Metrics
- MIMD class HPC Architecture
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architectures



# Pipeline Structures

- Partitioning of functional unit into a sequence of stages
  - Execution time of each stage is < that of the original unit
  - Total time through sequence of stages is usually > that of the original unit
- Pipeline permits overlapping of multiple operations
  - At any one time: each stage is performing different operation
  - # of operations being performed in parallel = # stages
- Performance
  - Pipeline increments at clock rate of slowest pipeline stage
  - Response time for an operation is product of # stages and clock cycle time
  - Throughput = clock rate
    - i.e. one operation result per clock cycle of pipeline
- Pipeline structures employed in many parts of a computer architecture
  - to enable high throughput in the presence of high latency
  - enable faster clock rates

# Pipeline: Concepts



Where :

- $T_c$  is the Logic Latency
- $T_p$  is the aggregated pipeline latency
- $t_p$  is the latency for each pipelined step

$$t_p \ll T_c$$

$$T_p = N \times t_p$$

$$T_c < T_p$$

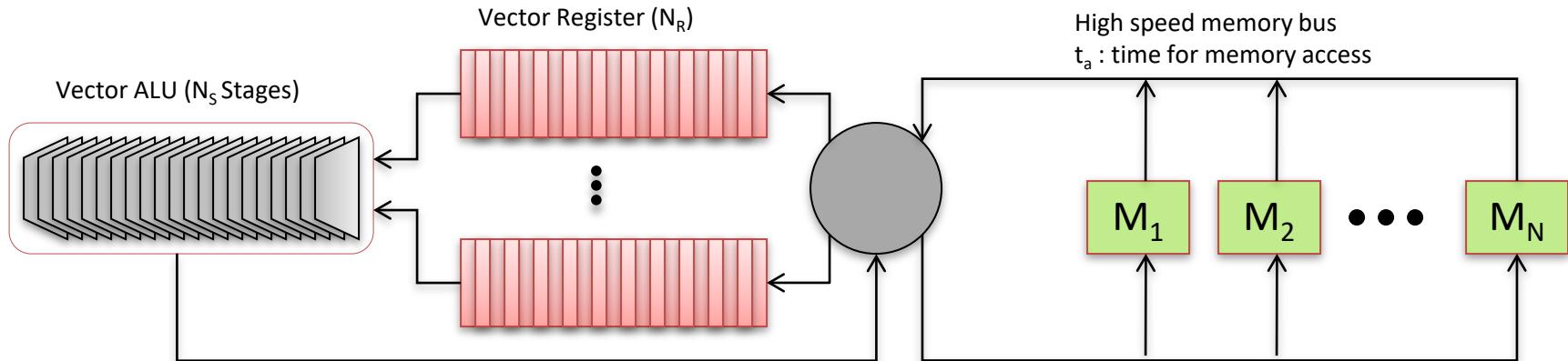
$$Perf_c = \frac{1}{T_c}$$

$$Perf_p = \frac{1}{t_p}$$

# Vector Processors

- Supports fine grained data parallel semantics
  - Many instances of same operation performed concurrently under same control element
  - Operates on vector data structures rather than single scalar values
  - Vector-scalar operations
    - Scale a vector by a scalar factor (multiply each vector element by scalar)
  - Inter-vector operations
    - e.g., Pair wise multiplies
  - Intra-vector operation
    - Reduction operators
      - e.g., sum all elements of a vector
- Exploits pipeline structure
  - Arithmetic units
  - Vector registers
  - Overlap of memory banks access cycles
  - Overlap of communication with computation
- Limited scaling – upper bound on number of pipeline stages

# Vector Pipeline Architecture



$$\text{Ideal Vector Performance} = \frac{1}{t_c}$$

$$T_{MV} = t_s + \sum_{a=1}^N t_a$$

$$\text{Achieved Vector Performance} = \frac{N_R}{(N_s + N_R) \times t_c}$$

$$P_M = \frac{N}{T_{MV}}$$

$$N_{\frac{1}{2}} : N_s := N_R$$

$$Perf_R = \frac{N_R}{(N_R + N_s) \times t_c}$$

$$= \frac{N_R}{2 \times (N_R) \times t_c} = \frac{1}{2 \times t_c}$$

Where :

- $t_a$  is the time for memory access
- $t_s$  is the startup time
- $T_{MV}$  is the combined time for Memory Vector
- $P_M$  is the memory performance
- $t_c$  is the ALU clock time of each step

# Cray 1

- First announced in 1975-6
- 80 MHz Clock rate
- Theoretical peak performance (160 MIPS), average performance 136 megaflops, vector optimized peak performance 150 megaflops
- 1-million 64 bit words of high speed memory
- Manufactured by Cray Research Inc.
- First Customer was National Center for Atmospheric Research (NCAR) for 8.86 million dollars.



src : <http://en.wikipedia.org/wiki/Cray-1>

The Cray 1 System



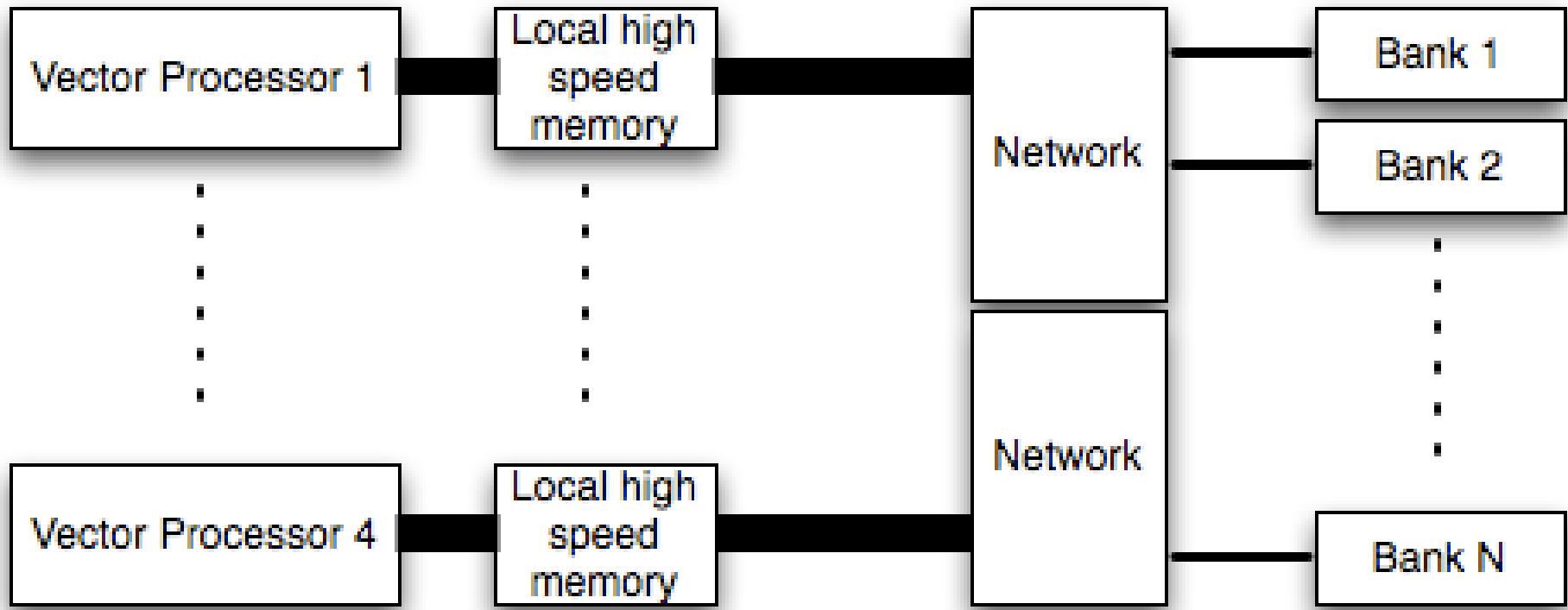
Cray 1 logic boards



# Parallel-Vector-Processors: PVP

- Combines strengths of vector and MPP
  - Efficiency of vector processing
    - Capability computing
  - Scalability of massively parallel processing
    - Capacity and cooperative computing
- Two levels of parallelism
  - Ultra fine grain vector parallelism with vector pipelining
  - Medium to coarse-grain processor
- Memory model
  - Alternative ways of organizing memory & address space
  - Distributed memory
    - Shared memory within node of multiple vector processors
    - Fragmented or decoupled address space between nodes
  - Partitioned global address space
    - Globally accessible address space
    - No cache coherence between nodes

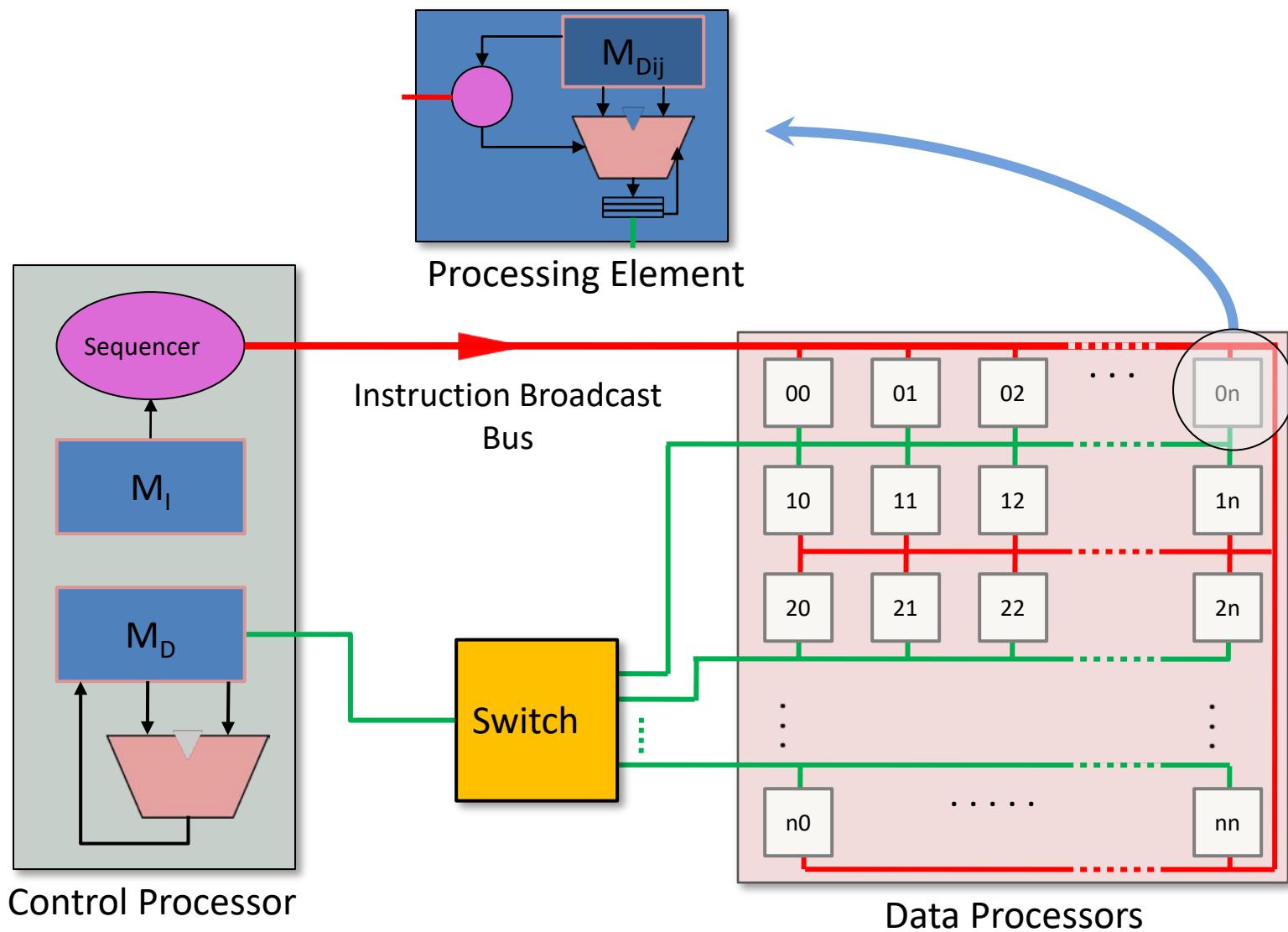
# PVP (e.g. Cray- XMP)



# SIMD Array

- SIMD semantics
  - Single Instruction stream Multiple Data stream
  - Data set partitioned in to blocks upon which
    - One or two dimensions (vectors or matrices)
  - Each data block is processed separately
  - Each data block is controlled by same instruction sequence
  - Data exchange cycle
- SIMD Parallel Structure
  - Node Array of arithmetic units, each coupled to local memory
  - Interconnect network for global data exchange
  - Single controller to issue instructions to array nodes
    - Early systems broadcast one instruction at a time
    - Modern systems point to sequence of cached instructions
- SPMD
  - Single Program Multiple Data Stream
  - Microprocessor based system where each node runs same program

# Simplified SIMD Diagram



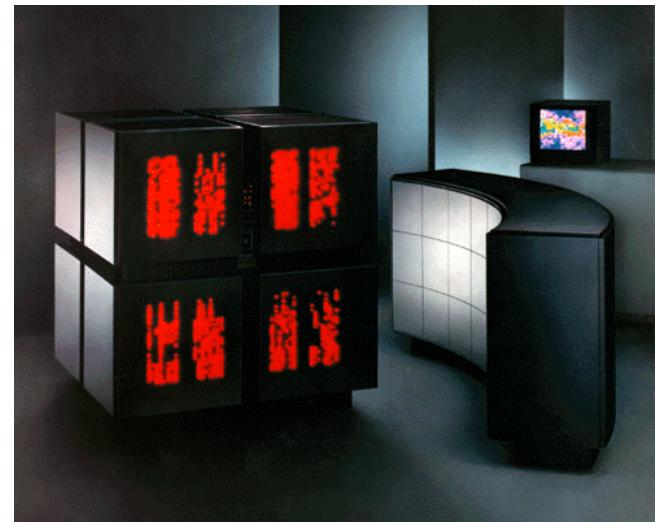
# **CM- 2**

## **CM-2 General Specifications :**

- Processors 65,536
- Memory 512 Mbytes
- Memory Bandwidth 300Gbits/Sec
- I/O Channels 8 Capacity per Channel 40 Mbytes/Sec Max.
- Transfer Rate 320 Mbytes/Sec
- Performance in excess of 2500 MIPS
- Floating Point performance in excess of 2.5 GFlops

## **DataVault Specifications :**

- Storage Capacity 5 or 10 Gbytes
- I/O Interfaces 2 Transfer Rate,
- Burst 40 Mbytes/Sec Max.
- Aggregate Rate 320 Mbytes/Sec
- Originated at MIT, by Danny Hillis
- Commercialized at Thinking Machines Corp.



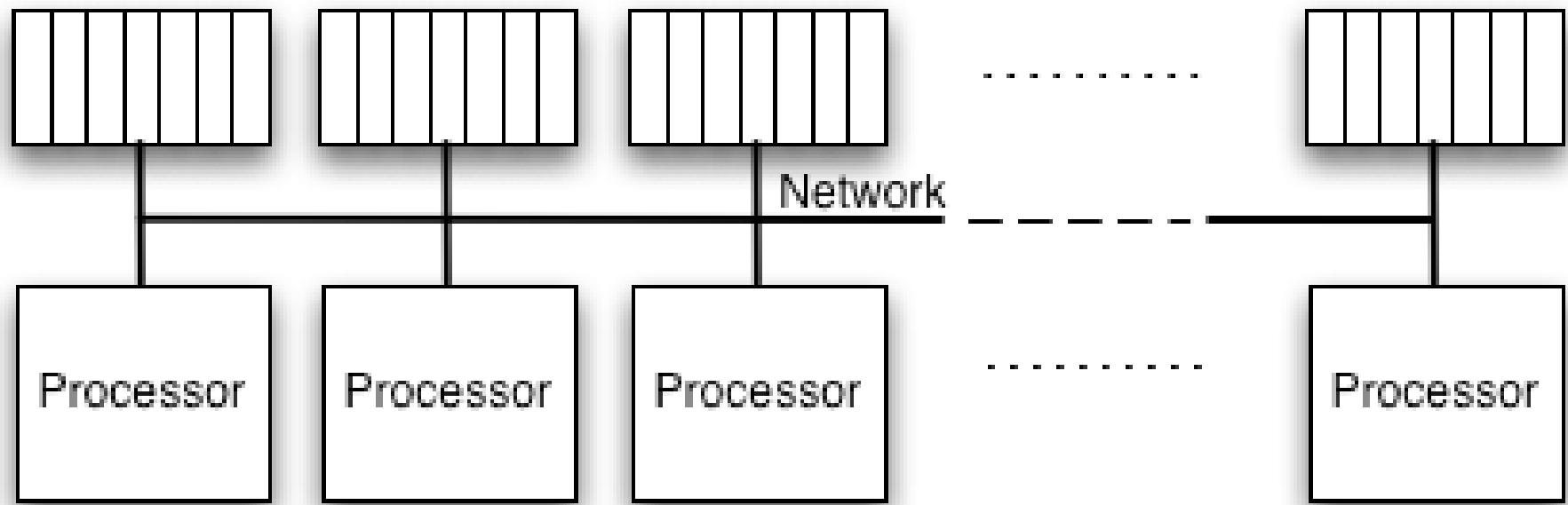
src : <http://www.svisions.com/sv/cm-dv.html>

# Introduction to SMP

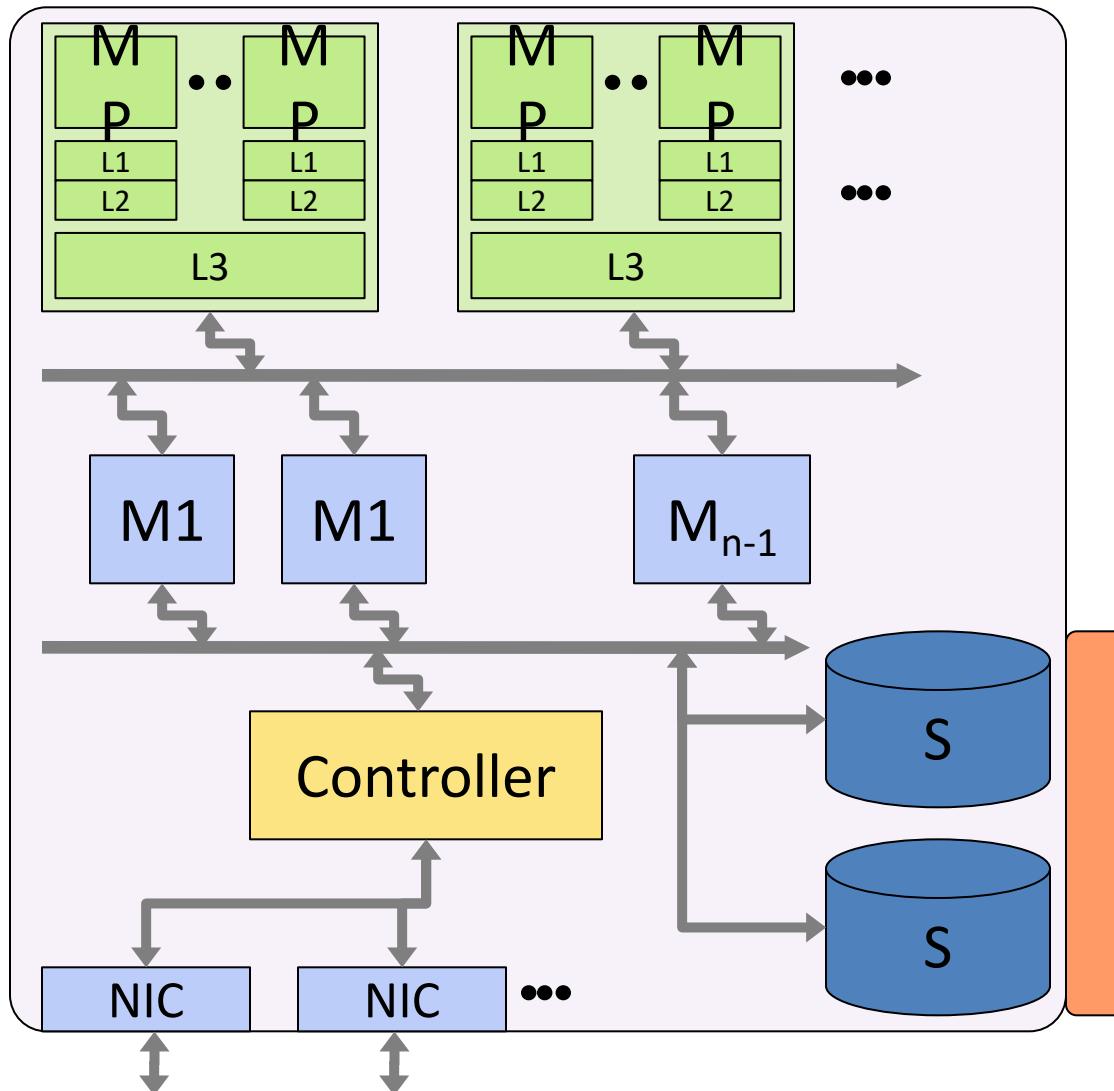
- Symmetric Multiprocessor
- Building block for large MPP
- Multiple processors
  - 2 to 32 processors
  - Now Multicore
- Uniform Memory Access (UMA) shared memory
  - Every processor has equal access in equal time to all banks of the main memory
- Cache coherent
  - Multiple copies of variable maintained consistent by hardware

# SMP- UMA

Memory Banks



# SMP Node Diagram



Legend :

MP : MicroProcessor

L1,L2,L3 : Caches

M1.. : Memory Banks

S : Storage

NIC : Network Interface Card

PCI-e

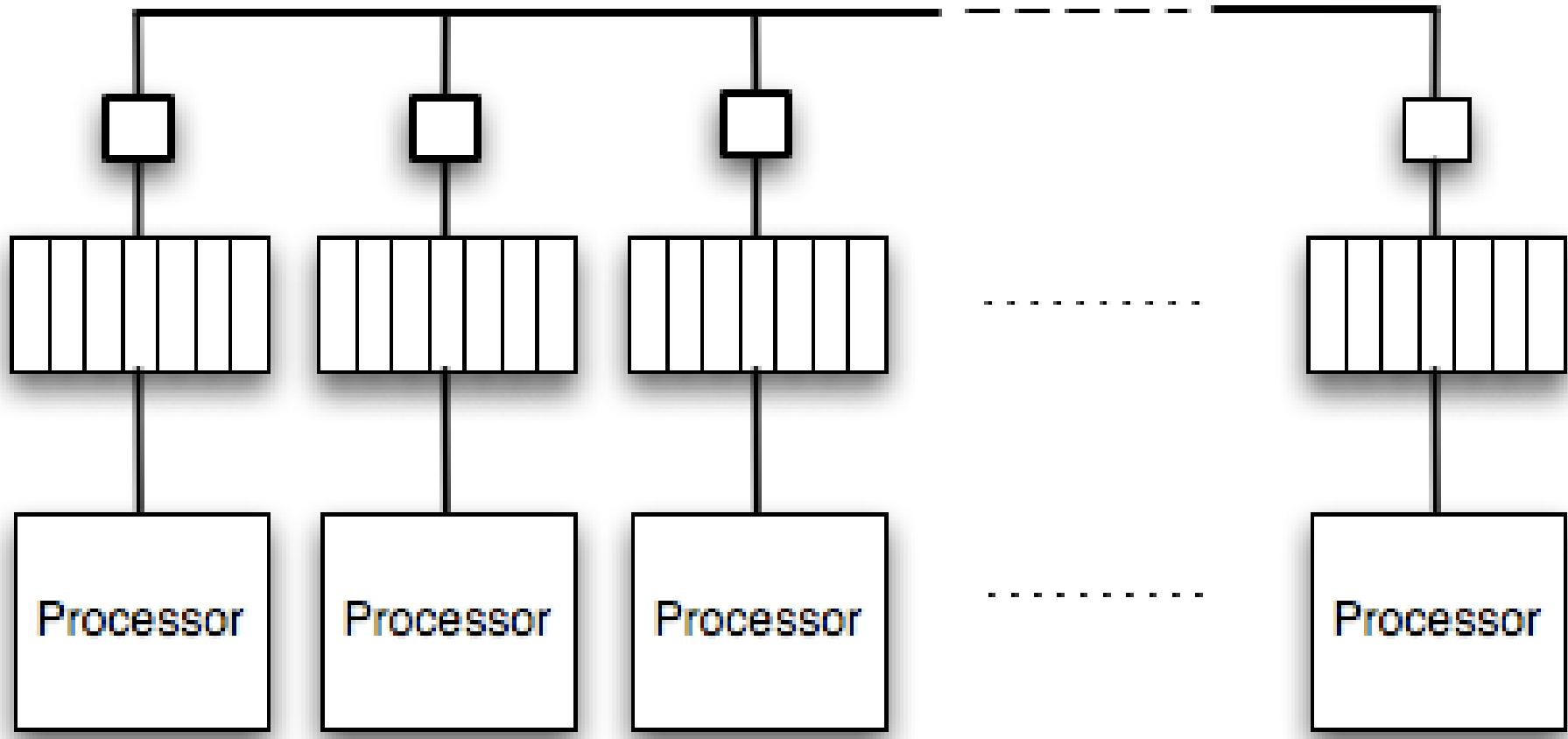
JTAG

Ethernet

Peripherals

USB

# DSM- NUMA

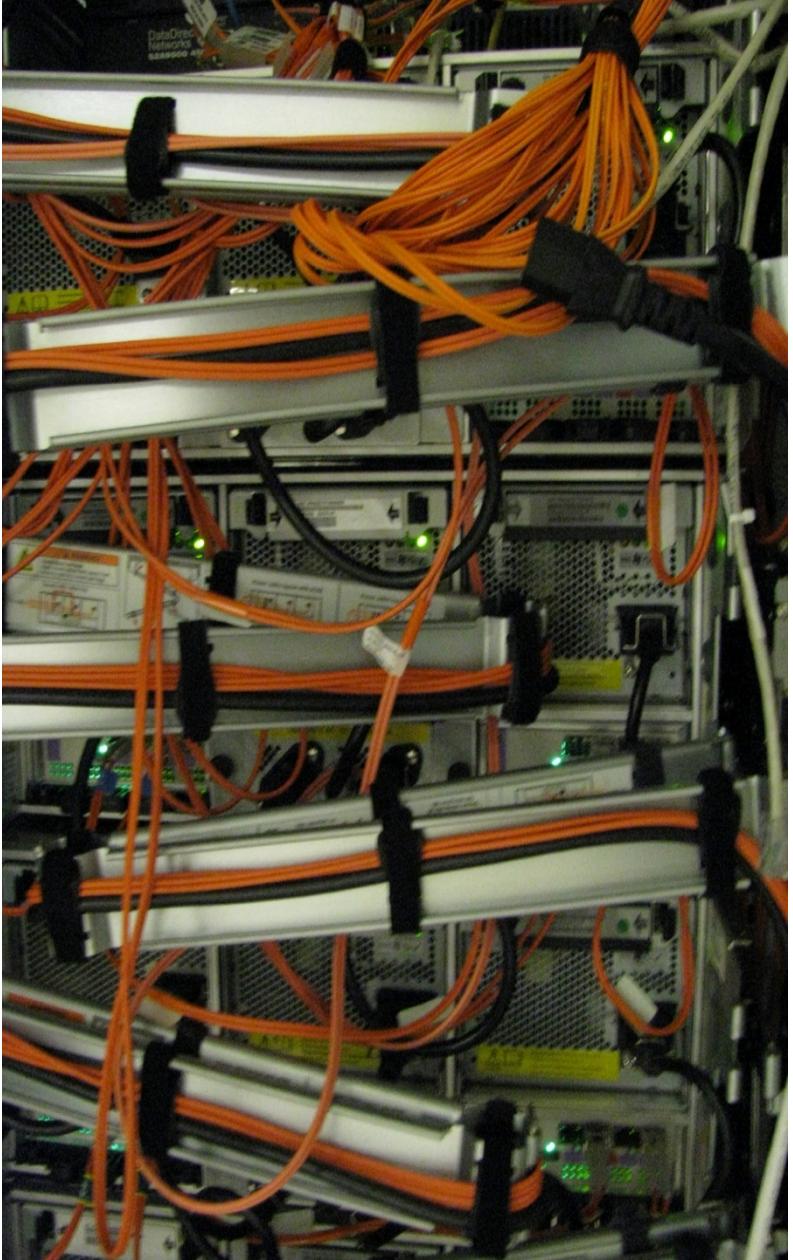


# Challenges to Computer Architecture

- Expose and exploit extreme fine-grain parallelism
  - Possibly multi-billion-way (for Exascale)
  - Data structure-driven (use meta-data parallelism)
- State storage takes up much more space than logic
  - 1:1 flops/byte ratio infeasible
  - Memory access bandwidth is the critical resource
- Latency
  - can approach a million cycles (10,000 or more cycles, typical)
  - All actions are local
  - Contention due to inadequate bandwidth
- Overhead for fine grain parallelism must be very small
  - or system can not scale
  - One consequence is that global barrier synchronization is untenable
- Power consumption
- Reliability
  - Very high replication of elements
  - Uncertain fault distribution
  - Fault tolerance essential for good yield
- Design complexity
  - Impacts development time, testing, power, and reliability

# Topics

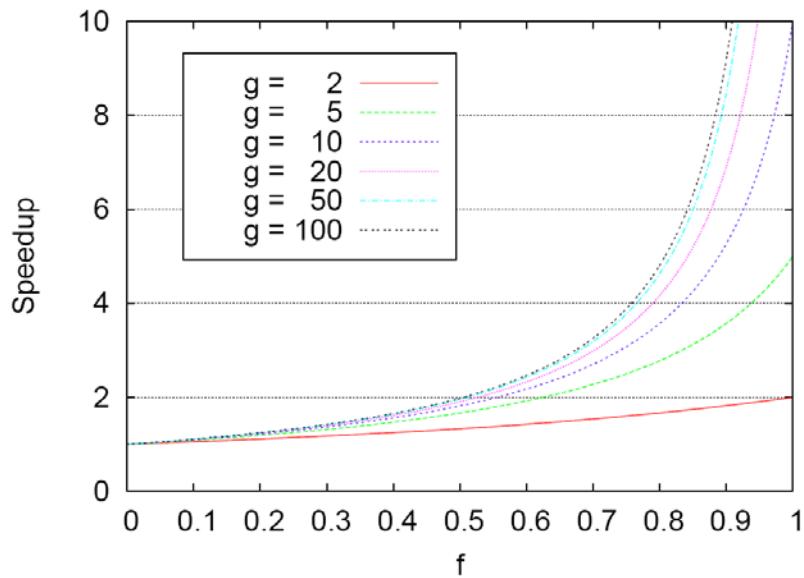
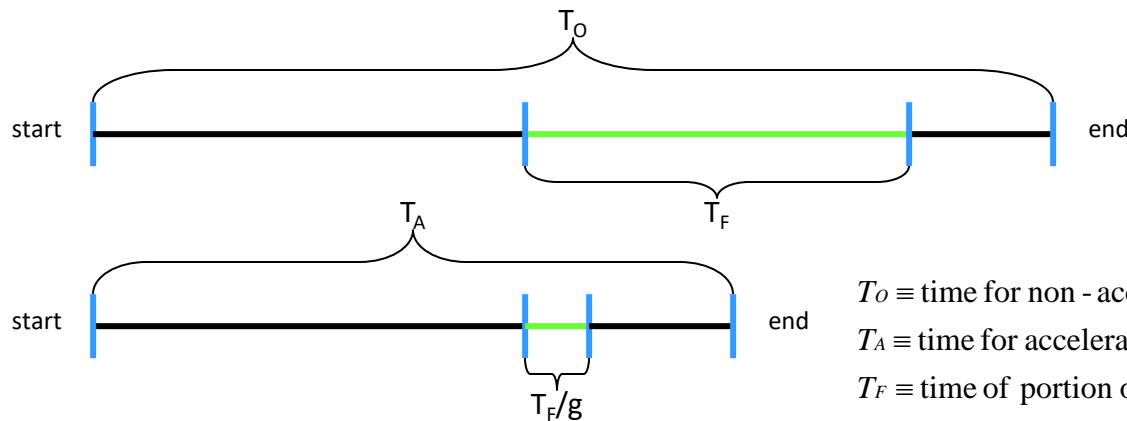
- The HPC System Stack
- What is Computer Architecture
- Review Performance Factors and Metrics
- MIMD class HPC Architecture
  - An Introduction to Shared Memory Multiprocessors
  - Coarse-grained MIMD Processing – MPPs
- SIMD class HPC Architecture
  - Very Fine-grained Vector Processing and PVPs
  - SIMD array and SPMD
- Current generation multicore and heterogeneous architectures



# Multi-Core

- Motivation for Multi-Core
  - Exploits increased feature-size and density
  - Increases functional units per chip (spatial efficiency)
  - Limits energy consumption per operation
  - Constrains growth in processor complexity
- Challenges resulting from multi-core
  - Relies on effective exploitation of multiple-thread parallelism
    - Need for parallel computing model and parallel programming model
  - Aggravates memory wall
    - Memory bandwidth
      - Way to get data out of memory banks
      - Way to get data into multi-core processor array
    - Memory latency
    - Fragments L3 cache
  - Pins become strangle point
    - Rate of pin growth projected to slow and flatten
    - Rate of bandwidth per pin (pair) projected to grow slowly
  - Requires mechanisms for efficient inter-processor coordination
    - Synchronization
    - Mutual exclusion
    - Context switching

# 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

$$S = T_o / T_A$$

$$f = T_F / T_o$$

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

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

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

# Commodity Clusters

# What is a Commodity Cluster?

- It is a distributed/parallel computing system
- It is constructed entirely from commodity subsystems
  - All subcomponents can be acquired commercially and separately
  - Computing elements (nodes) are employed as fully operational standalone mainstream systems
- Two major subsystems:
  - Compute nodes
  - System area network (SAN)
- Employs industry standard interfaces for integration
- Uses industry standard software for majority of services
- Incorporates additional middleware for interoperability among elements
- Uses software for coordinated programming of elements in parallel



## BEOWULF

JPL

PC Clusters Deliver Supercomputer Performance at Mass-Market Price

10 Gflops Scale Performance

16 - 128 CPUs

Intel Pentium Pro (200MHz) Processors

128 Mbytes Memory and 3 Gbytes Disk per Node

100 Mbit/s Ethernet with Gigabit Options

Linux O/S with MPI, PVM, & LAM

Total Cost: \$2400 per Node

## The Beowulf Project

Supercomputer Performance at PC Prices



Architecture  
Nodes  
Shared Memory  
Network  
Software  
Performance

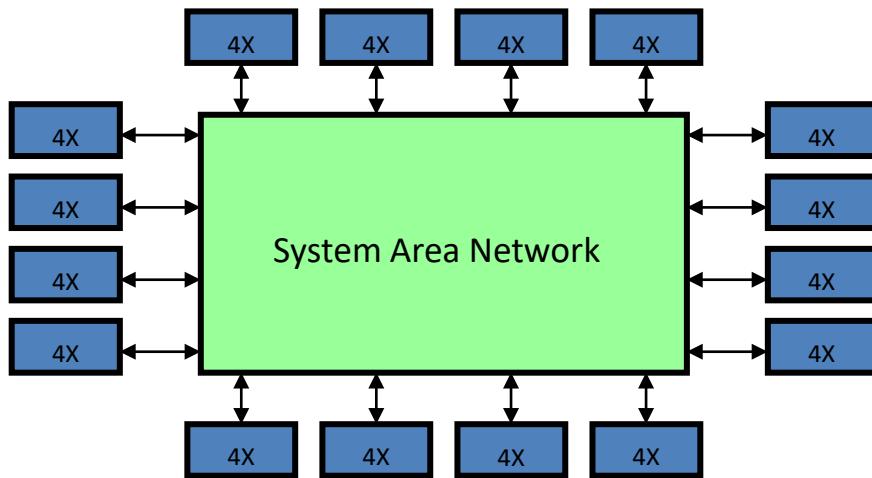


NÆGLING  
THE SWORD OF BEOWULF

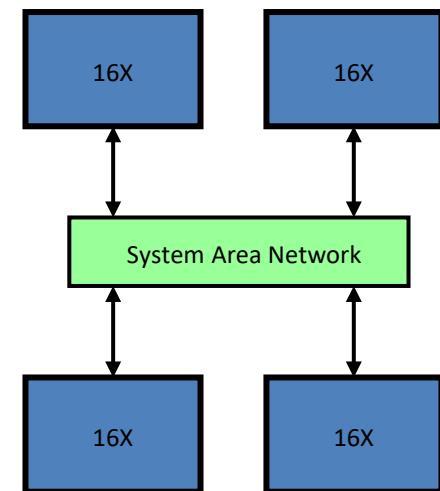


# Commodity Clusters vs “Constellations”

- An ensemble of  $N$  nodes each comprising  $p$  computing elements
- The  $p$  elements are tightly bound shared memory (e.g., smp, dsm)
- The  $N$  nodes are loosely coupled, i.e., distributed memory

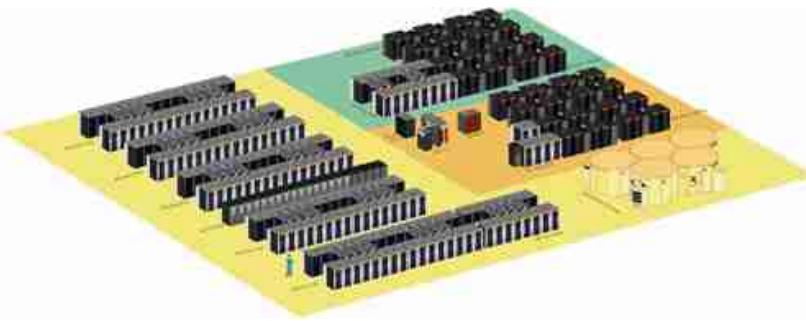


- $p$  is greater than  $N$
- Distinction is which layer gives us the most power through parallelism



64 Processor  
Constellation

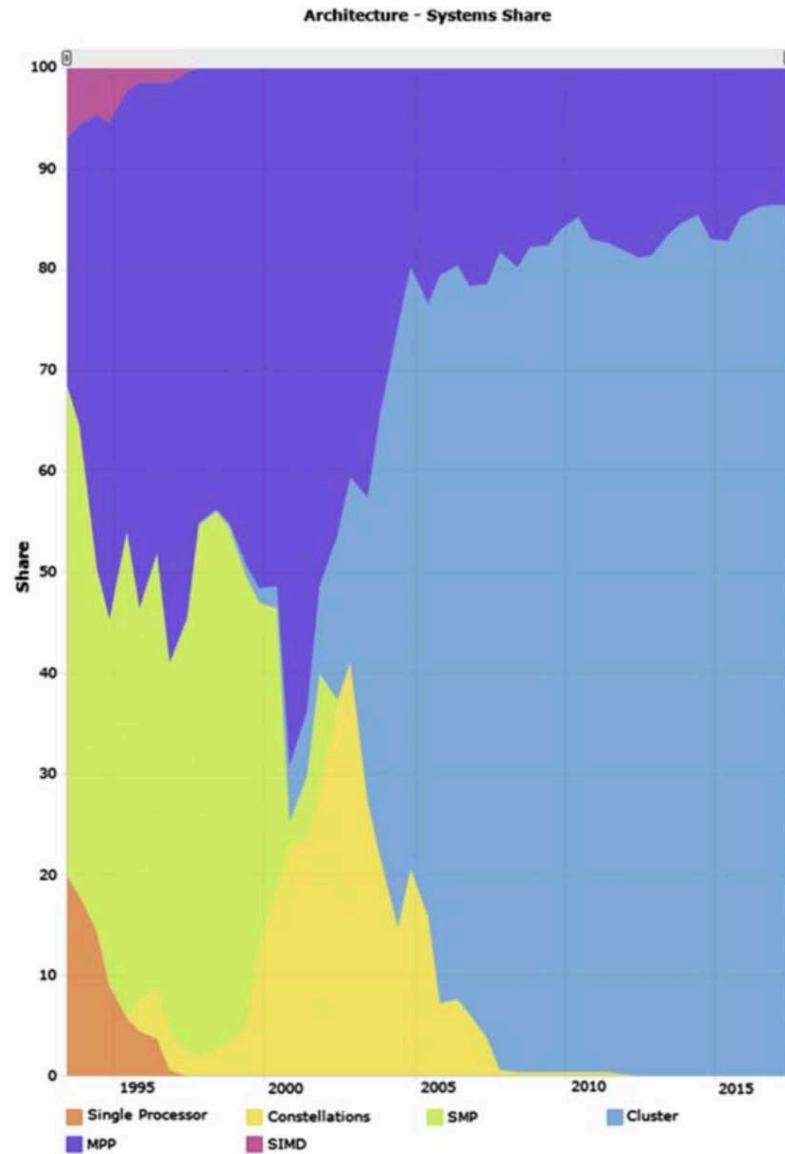
# Columbia



- NASA's largest computer
- NASA Ames Research Center
- A Constellation
  - 20 nodes
  - SGI Altix 512 processor nodes
  - Total: 10,240 Intel Itanium-2 processors
- 400 Terabytes of RAID
- 2.5 Petabytes of silo farm tape storage



# Clusters Dominate Top-500

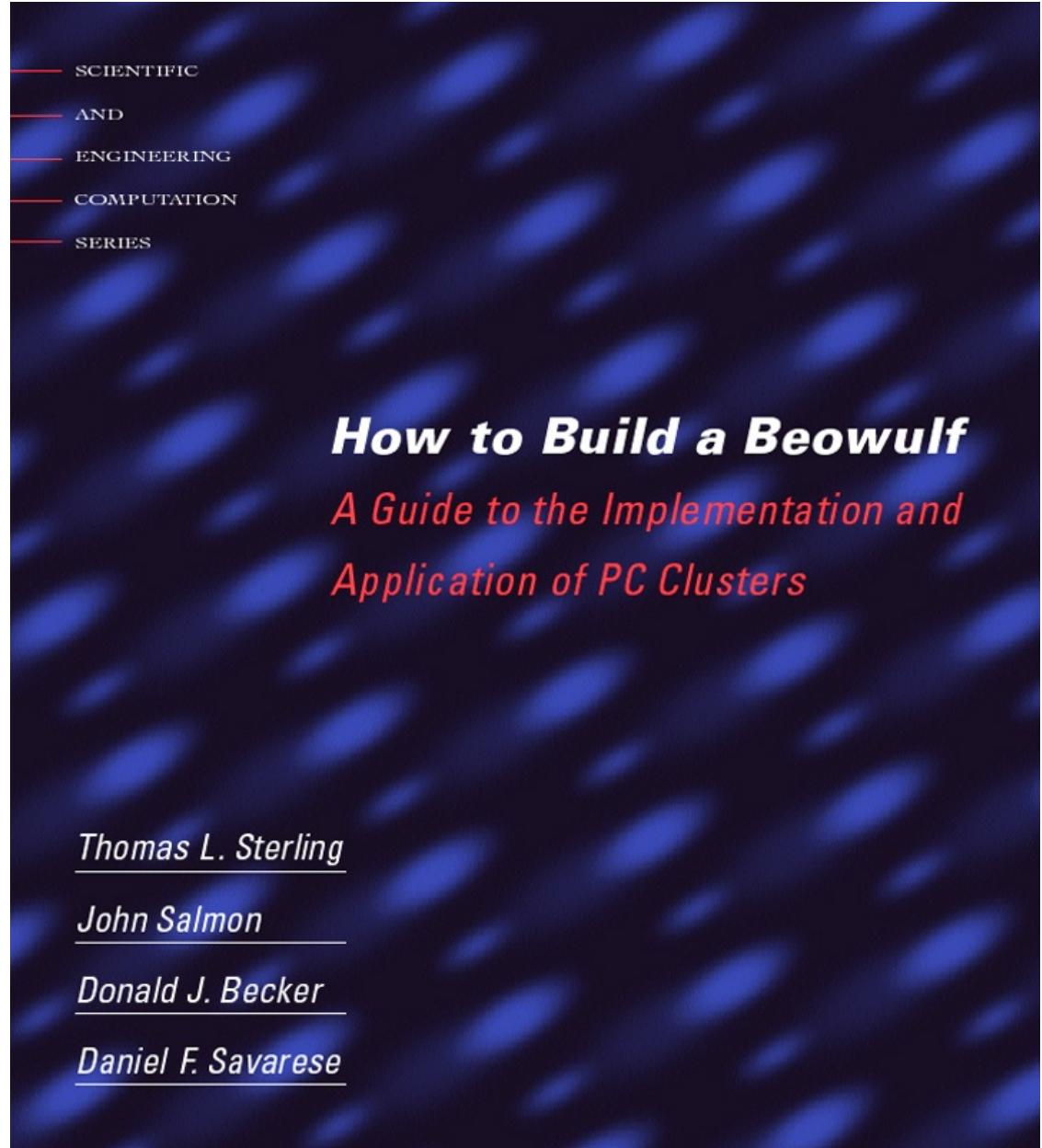


The dominant system architecture classes comprising the fastest 500 computers over the last 24 years.

# Why are Clusters so Prevalent

- Excellent performance to cost for many workloads
  - Exploits economy of scale
    - Mass produced device types
    - Mainstream standalone subsystems
  - Many competing vendors for similar products
- Just in place configuration
  - Scalable up and down
  - Flexible in configuration
- Rapid tracking of technology advance
  - First to exploit newest component types
- Programmable
  - Uses industry standard programming languages and tools
- User empowerment
  - Low cost, ubiquitous systems
  - Programming systems make it relatively easy to program for expert users

1st printing: May, 1999  
2nd printing: Aug. 1999  
MIT Press





The 1996 1 Gigaflops Beowulf cluster.

**Table 3.1** Overview of Components of Several Commercially Available Commodity Clusters

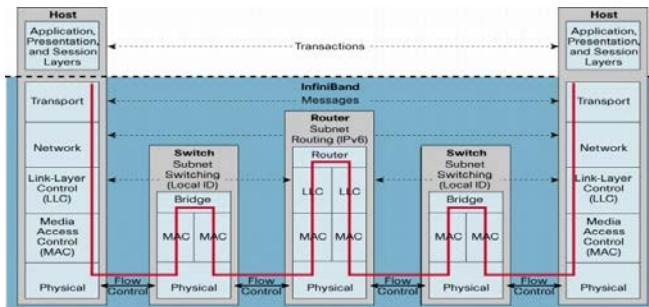
Machine	Network	Processor	Cores per Node	Memory Capacity	Blades	Vendor	Secondary Storage	Nodes per Rack
SuperMUC	Infiniband-FDR (41.25 Gb/s) Mellanox	Sandy Bridge—EP Intel Xeon E5-2680 8C, 2.7 GHz (Turbo 3.5 GHz)	16	32 GB/node	Yes	IBM/ Lenovo	15 PB (scratch) 3.5 PB (home)	512
Mistral	Infiniband-FDR	Xeon E5-2680v3 12C 2.5 GHz/ E5-2695v4 18c 2.1 GHz	24	64 GB/node	No	Bull, Atos		18
Cray CS-Storm	Infiniband-FDR	Xeon E5-2660v2 10C 2.2 GHz	20	Up to 1024 GB/node	No	Cray		23
Stampede	Infiniband-FDR (56 Gbps)	Xeon E5-2680 8C 2.7 GHz	16	32 GB/node	No	Dell	14 PB (shared) 1.6 PB (local aggregate)	40
HPC4 HP POD	Infiniband-FDR	Xeon E5-2697v2 12C 2.7 GHz	24		Yes	Hewlett—Packard	1.8 PB (shared) 0.75 PB (midterm shared) 1.5 PB (local aggregate)	160

# Programming on Clusters

- Several ways of programming application on clusters
  - Throughput – jobstream
  - Decoupled Work Queue Model – SPMD for parameter studies
  - Communicating Sequential Processes (CSP)
  - Multi threaded
- Throughput: job stream
  - PBS, Maui
- Decoupled Work Queue Model : SPMD, e.g. parametric studies
  - Condor
- Communicating Sequential Processes
  - Message passing
  - Distributed memory
  - Global barrier synchronization
  - e.g., MPI
- Multi threaded
  - Limited to intra-node programming
  - Shared memory
  - e.g., OpenMP

# Some Node Interconnect Options

- Current Generation
  - Gigabit Ethernet (~1000 Mb/s)
  - 10 Gigabit Ethernet
  - 40 Gigabit Ethernet and 100 Gigabit Ethernet (100GbE)  
standards are in draft as of 2009
  - Infiniband (IBA)
    - High Performance: 10 - 20 Gbps
    - Low latency: 1.2 microseconds
    - Copper interconnects
    - High availability - IEEE 802.3ad Link Aggregation / Channel Bonding
- Previous Generation
  - Fast Ethernet (~100 Mb/s)
  - Myricom's Myrinet-2000 (~1600 Mb/s)
  - SCI (~4000 Mb/s)
  - OC-12 ATM (~622 Mb/s)
  - Fiber Channel (~100 MB/s)
  - USB (12 Mb/s)
  - Firewire (IEEE 1394 400 Mb/s)





# Schedulers: PBS

Workload management system – coordinates resource utilization policy and user job requirements

- Multi users, Multi jobs, Multi nodes
- Both Open Source and Commercially supported
- Functionality
  - Manages parallel job execution
  - Interactive and batch cross system scheduling
  - Security and access control lists
  - Dynamic distribution and automatic load-leveling of workload
  - Job and user accounting
- Accomplishments
  - Runs on all Unix and Linux platforms
  - Supports MPI
  - First release 1995
  - 2000 sites registered, 1000 people on the mailing list
  - PBSPro sales at >5000 cpu's

# MPI Software

- Community wide standard process
  - Leveraged experiences with NX, PVM, P4, Zipcode, others
- Dominant programming model for clusters
- Multiple implementations both OSS and commercial (MPI Soft Tech)
  - All of MPI-1
  - MPI I./O
  - All of MPI-2
  - MPI-3 under development
- Functionality
  - Message passing model for distributed memory platforms
  - Support for truly scalable operations (1000s nodes)
    - Rich set of collective operations (gathers, reduces, scans, all to all)
    - Scalable one sided operations (fence barrier synchronization, group-oriented synchronization)
  - Dynamic processes (2) to spawn, disconnect etc. with scalability
- MPICH-2 entirely new rewrite
- OpenMPI includes fault tolerant capability



# Compilers & Debuggers

- Compilers :
  - Intel C/ C++ / Fortran
  - PGI C/ C++ / Fortran
  - GNU C / C++ / Fortran
- Libraries :
  - Each compiler is linked against MPICH
  - Mesh/Grid Partitioning software : METIS etc.
  - Math Kernel Libraries (MKL)
  - Intel MKL, AMD MKL, GNU Scientific Library (GSL)
  - Data format libraries : NetCDF, HDF 5 etc
  - Linear Algebra Packages : BLAS, LAPACK etc
- Debuggers
  - gdb
  - Totalview
- Performance & Profiling tools :
  - PAPI
  - TAU
  - Gprof
  - perfctr

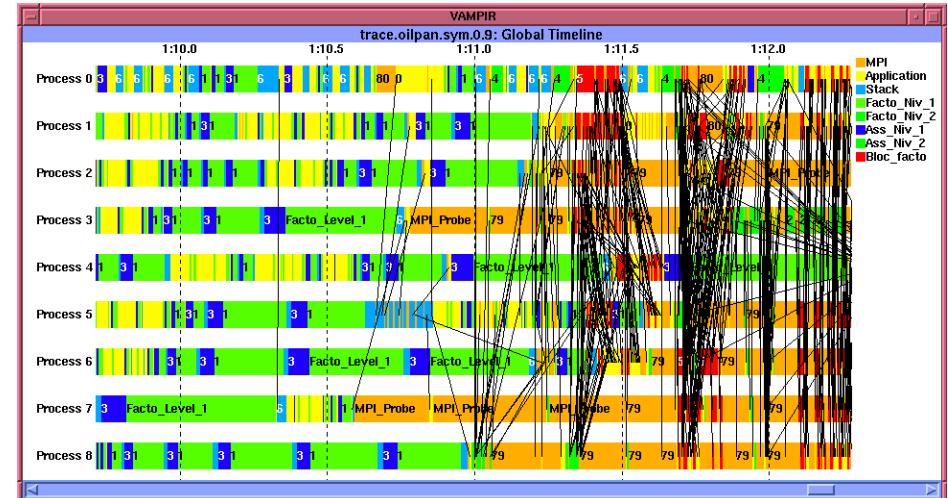


# Distributed File Systems

- A distributed file system is a file system that is stored locally on one system (server) but is accessible by processes on many systems (clients).
- Multiple processes access multiple files simultaneously.
- Other attributes of a DFS may include :
  - Access control lists (ACLs)
  - Client-side file replication
  - Server- and client- side caching
- Some examples of DFSees:
  - NFS (Sun)
  - AFS (CMU)
  - PVFS (Clemson, Argonne), OrangeFS
  - Lustre (Sun)
  - GPFS (IBM)
- Distributed file systems can be used by parallel programs, but they have significant disadvantages :
  - The network bandwidth of the server system is a limiting factor on performance
  - To retain UNIX-style file consistency, the DFS software must implement some form of locking which has significant performance implications

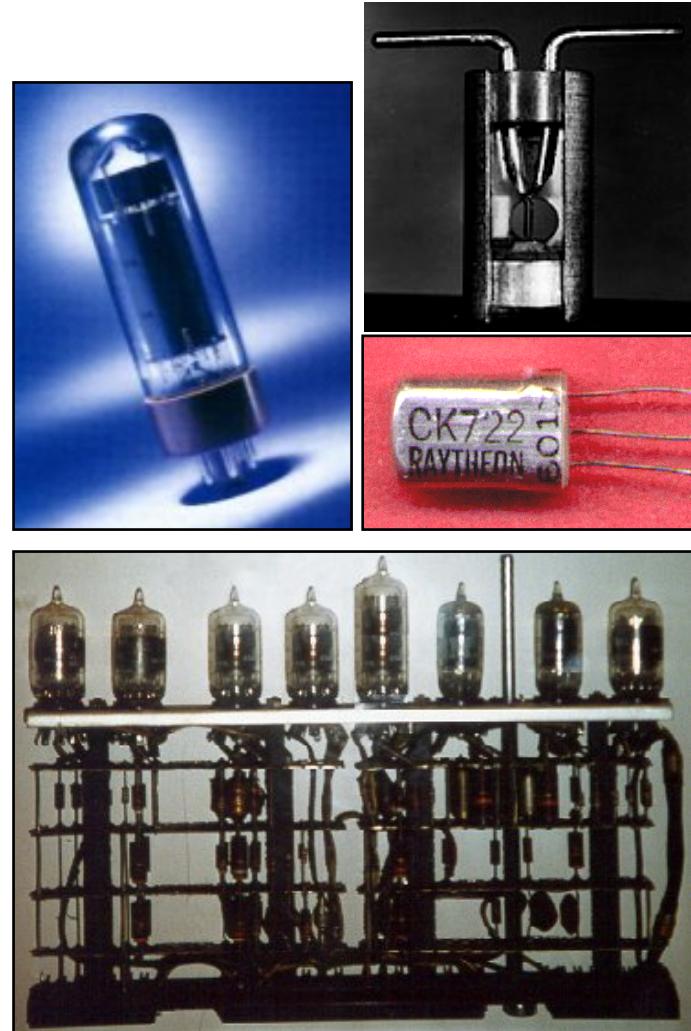
# Measuring Performance on Clusters

- Ways of measuring performance
  - Wall clock time
  - Benchmarks
  - Processor efficiency factors
  - Scalability
  - MPI communications and synchronization overhead
  - System operations
- Tools
  - PAPI
  - Tau
  - Ganglia
  - Many others



# Open Source Software

- Evolution of PC Clusters has benefited from Open Source Software
- Early examples
  - Gnu compiler tools, FreeBSD, Linux, PVM
- Advantages
  - Provides shared infrastructure – avoids duplication of effort
  - Permits wide collaborations
  - Facilitates exploratory studies and innovation
- Free software is not necessarily OSS
- Business model in state of flux: how to fund free deliverables
- Important synergy between OSS standard infrastructure software and proprietary ISV target-specific software:
  - OSS provides common framework
  - For-profit software provides incentive and resources



# The History of Linux

- Started out with Linus' frustration on available affordable operating systems for the PC
- He put together a rudimentary scheduler, and later added on more features until he could bootstrap the kernel (1991).
- The source was released on the internet in hope that more people would contribute to the kernel
- GCC was ported, a C library was added and a primitive serial and tty driver code
- Networks, file systems were added
- Slackware
- RedHat
- Extreme Linux

# Linux Distributions

Alphanet Linux	Embedix	Linux by Linux	Platinum Linux	Vine Linux
Alzza Linux	Enoch	Linux GT Server Edition	Power Linux	White Dwarf Linux
Andrew Linux	Eonova Linux	Linux Mandrake	Progeny Debian	Whole Linux
Apokalypse	ESware	Linux MX	Project Freesco	WinLinux 2000
Armed Linux	Etlinux	LinuxOne	Prosa Debian	WorkGroup Solutions
ASPLinux	Eurielec Linux	LinuxPPC	Pygmy Linux	Linux Pro Plus
Bad Penguin	FinnixFloppi	Gentoo	Red Flag Linux	Xdenu
Bastille Linux	Linux	LinuxPPP	Red Hat Linux	Xpresso Linux 2000
Best Linux	Gentus Linux	LinuxSIS	Redmond Linux	XTeam Linux
BlackCat Linux	Green Frog Linux	LinuxWare	Rock Linux	Yellow Dog Linux
Blue Linux	Halloween Linux	Linux-YeS	RT-Linux	Yggdrasil Linux
Bluecat Linux	Hard Hat Linux	LNX System	Scrudge Ware	ZiiF Linux
BluePoint Linux	HispaFuentes	Lunet	Secure Linux	ZipHam
Brutalware	HVLinux	LuteLinux	Skygate Linux	ZipSlack
Caldera OpenLinux	Icepack	LST	Slacknet Linux	
Cclinux	Immunix	Mastodon	Slackware	
ChainSaw Linux	OSIndependence	MaxOS&trade;	Slinux	
CLECILeINUX	InfoMagick Workgroup	MIZI Linux OS	SOT Linux	
Conectiva	Server	MkLinux	Spiro	
CoolLinux	Ivrix	MNIS Linux	Stampede Linux	
Coyote Linux	ix86 Linux	MicroLinux	Storm Linux	
Corel	JBLinux	Monkey Linux	S.u.SE	
COX-Linux	Jurix Linux	NeoLinux	Thin Linux	
Darkstar Linux	Kondara	Newlix OfficeServer	TINY Linux	
Debian Definite	Krud	NoMad Linux	Trinux	
Linux	KW Linux	Ocularis	Trustix Secure Linux	
deepLINUX	KSI Linux	Open Kernel Linux	TurboLinux	
Delix	L13Plus	Open Share Linux	Turquaz	
Dlite (Debian Lite)	Laser5	OS2000	UltraPenguin	
DragonLinux	Leetnux	Peanut Linux	Ute-Linux	
Eagle Linux M68K	Lightening	PhatLINUX	VA-enhanced RedHat Linux	
easyLinux	Linpus Linux	PingOO	VectorLinux	
Elfstone Linux	Linux Antarctica	Plamo Linux	Vedova Linux	

# Benchmarking

# Benchmarking

- *Benchmarking* is a way to empirically measure the performance of a supercomputer.
- A *benchmark* provides some standardized type of workload that may vary in size or input data set.
- Benchmarks provide as output a performance metric that can be used for comparing the performance between different supercomputers based on that benchmark's specific workload.
- Some example performance metrics: floating point operations per second (Flops), traversed edges per second (Teps), Giga updates per second (Gups), degrees of freedom per second (Dofs).
- Computational benchmark workloads come in two types: synthetic, where workloads are designed and created to impose a load on a specific component in the system; and application, where the workload is derived from a real-world application.

# Historical Benchmarking Efforts

- Whetstone benchmark named after the Whetstone village in Leicestershire England
- This benchmark, first released in 1972, consisted of multiple programs that created synthetic workloads for evaluating Kilo Whetstone Instructions per second.
- Benchmark is floating point intensive.
- In 1980 it was updated to report floating point operations per second.
- Can still be downloaded:  
[www.netlib.org/benchmark](http://www.netlib.org/benchmark)

# Historical Benchmarking Efforts

- Dhrystone -- a benchmark with a standardized synthetic computing workload for measuring integer performance.
- Name reflects its function as the integer counterpart to Whetstone.
- This benchmark, like Whetstone, would become an industry standard.
- Superseded by the SPECint suite.
- Still downloadable: [www.netlib.org/benchmark](http://www.netlib.org/benchmark)

# LINPACK

- The genesis of one of the most widely used benchmarks in supercomputing is the Linpack benchmark introduced by Jack Dongarra in 1979 and based on the Linpack linear algebra package developed by Jack Dongarra, Jim Bunch, Cleve Moler, and Gilbert Stewart .
- The Linpack linear algebra package has been superseded by LAPACK but the Linpack benchmark is going strong.
- The Linpack benchmark employs a workload that solves a dense system of linear equations. That is, it solves for  $x$  in
$$Ax=b$$

where  $b$  and  $x$  are vectors of length  $n$  and  $A$  is an  $n \times n$  matrix with very few or no zero elements.

# LINPACK

- The original Linpack benchmark solved matrices with  $n=100$  and was written for serial computation.
- No changes to the source code were allowed; only optimizations achieved through compiler flags were allowed.
- A second iteration of the benchmark used matrices with  $n=1000$  and allowed user modifications to the factorization and solver portions of the code. An accuracy bound on the final solution was also introduced.
- The third iteration of the benchmark, called the Highly Parallel Computing Linpack or HPL, allows variations in both the problem size and software and will run on a distributed memory supercomputer.
- This version of the benchmark is used to generate the Top500 list that is frequently used to rank supercomputers throughout the world.

Today there are a wide variety of general-purpose benchmarks used for evaluating the performance of supercomputers and supercomputing elements. Many are motivated by specific classes of application workloads.

Benchmark	Application Domain Workload	Aim	Parallelism	Characteristics
<b>HPL</b>	Dense linear algebra	Estimate system's effective Flops	MPI	Part of HPC Challenge Benchmark; used for Top500 list
<b>STREAM</b>	synthetic	Estimate sustainable memory bandwidth (GB/s)	None	Part of the HPC Challenge Benchmark
<b>RandomAccess</b>	Synthetic	Estimate system's effective rate of integer random updates of memory – reported as Giga Updates per second (GUPS)	MPI, OpenMP	Part of the HPC Challenge Benchmark
<b>HPCG</b>	Sparse linear algebra	Estimate system's effective Flops for those applications poorly represented by HPL	MPI+OpenMP	Used for HPCG list ranking
<b>SPEC CPU 2006</b>	Various	Estimate system's effective processor, memory, and compiler performance	None	Commercial
<b>HPGMG</b>	Geometric multigrid	Estimate system's effective evaluation of number of degrees of freedom/second (DOFS)	MPI+OpenMP +CUDA	Used for HPGMG list ranking. Comes in two flavors: finite element and finite volume
<b>IS</b>	Computational Fluid Dynamics	Estimate system's effective integer sort and random access performance	MPI, OpenMP	Part of the NAS Parallel Benchmarks
<b>Graph500</b>	Data intensive applications	Estimate system's effective traversed edges per second (TEPS) for a graph traversal	MPI, OpenMP	Used for the Graph500 list ranking.

# Key properties of an HPC Benchmark

A good benchmark is:

- Relevant and meaningful to the target application domain.
- Applicable to a broad spectrum of hardware architectures.
- Adopted both by users and vendors and enable comparative evaluation.

Good benchmarks also tend to be relatively short and use standard parallel programming API's

Benchmark	Approximate Line Count	Parallelism		Language	
		MPI	OpenMP	C	C++
HPL	26700	X		X	
STREAM	1500			X	
RandomAccess	5800	X	X	X	
HPCG	5700	X	X		X
IS	1150	X	X	X	
Graph500	1900	X	X	X	
HPGMG	5000	X	X	X	

Because a supercomputer's performance can be compared and ranked based entirely off of a single benchmark, multiple differing supercomputing performance rankings exist. Here are the top machines for four different lists: Top500, Top HPCG, Graph500, and Top HPGMG for June 2017

Benchmark	Supercomputer	Location	Performance Result	Cores
HPL	Sunway TaihuLight	Jiangsu, China	93.0 Petaflops	10,649,600
HPGC	K computer	Kobe, Japan	0.6027 Petaflops	705,024
Graph500	K computer	Kobe, Japan	38621.4 GTEPS	705,024
HPGMG	Cori	Berkely, CA, USA	859 GigaDOFS	632,400

A supercomputer that performs well on one benchmark may not do so well on another:

Supercomputer	Top500 List Ranking	Graph500 List Ranking	HPCG Ranking	HPGMG Ranking
Sunway Taihu Light	1	2	3	2
Tianhe-2	2	8	2	
K Computer	8	1	1	
Cori	6		6	1

# HPC Challenge Benchmark Suite

- Consists of seven different tests:
  - HPL: LU factorization of dense linear algebra
  - DGEMM : double precision matrix-matrix multiplication
  - STREAM : synthetic workload to measure sustainable memory bandwidth
  - PTRANS : parallel matrix transpose
  - RandomAccess : reports the rate of integer random updates of memory in Giga-updates per second (GUPS)
  - FFT : double precision complex 1-D discrete Fourier transform
  - b\_eff : reports the latency and bandwidth for several different communication patterns

# HPL

- Key workload algorithm is LU factorization
- Given a problem size  $n$ , HPL will perform  $O(n^3)$  floating point operations while only performing  $O(n^2)$  memory accesses. Consequently, HPL is not strongly influenced by memory bandwidth.
- HPL contains many possible variations in the way it is executed so that the best performing approach for a particular supercomputer can be found empirically.
- The user is also allowed to entirely replace the LU factorization and solver step reference implementation with an alternative implementation if so desired.
- Unlike the earlier versions of Linpack, there are no restrictions on problem size in HPL.

# Running HPL/Linpack

The parameter space for tuning HPL is very large so parameter inputs separated by a space on each line are run independently.

Lines 5-13 specify problem sizes, block size configurations, grid configurations, and residual thresholds.

Lines 14-31 specify algorithmic variations in HPL. HPL has a number of different algorithm options, including 6 different virtual panel broadcast topologies (line 23), a bandwidth reducing swap-broadcast algorithm (line 26), back substitution with look-ahead depth of one (line 24), and three different LU factorization algorithms (lines 21) among other options.

Tuning for these parameters on a specific supercomputer is a routine task with HPL.

## *Example HPL input file*

```
1 HPLinpack benchmark input file
2 Innovative Computing Laboratory, University of Tennessee
3 HPL.out
4 6
5 4
6 29 30 34 35
7 4
8 1 2 3 4
9 0
10 3
11 2 1 4
12 2 4 1
13 16.0
14 3
15 0 1 2
16 2
17 2 4
18 1
19 2
20 3
21 0 1 2
22 1
23 0
24 1
25 0
26 2
27 64
28 0
29 0
30 1
31 8
```

output file name (if any)  
device out (6=stdout,7=stderr,file)  
# of problems sizes (N)  
Ns  
# of NBs  
NBs  
PMAP process mapping (0=Row-,1=Column-major)  
# of process grids (P x Q)  
Ps  
Qs  
threshold  
# of panel fact  
PFACTs (0=left, 1=Crout, 2=Right)  
# of recursive stopping criterium  
NBMINS (>= 1)  
# of panels in recursion  
NDIVs  
# of recursive panel fact.  
RFACTs (0=left, 1=Crout, 2=Right)  
# of broadcast  
BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)  
# of lookahead depth  
DEPTHs (>=0)  
SWAP (0=bin-exch,1=long,2=mix)  
swapping threshold  
L1 in (0=transposed,1=no-transposed) form  
U in (0=transposed,1=no-transposed) form  
Equilibration (0=no,1=yes)  
memory alignment in double (> 0)

## *Example HPL output*

T/V	N	NB	P	Q	Time	Gflops
WR11C2R4	1000	80	2	2	0.09	7.694e+00
Ax-b  _oo/(eps*(  A  _oo*  x  _oo+  b  _oo)*N)=						0.0072510 ..... PASSED

# Running HPL/Linpack

- For a certain problem size  $N_{\max}$ , the cumulative performance in Gflops reaches its maximum value called  $R_{\max}$ .
- The  $R_{\max}$  value is what is reported for the Top500 list ranking supercomputers.
- Another interesting metric from the HPL benchmark is  $N_{1/2}$ .  $N_{1/2}$  is the problem size where the maximum performance achieved is  $R_{\max}/2$ .

# Running HPC Challenge Benchmark

HPC Challenge benchmark runs all seven benchmarks at once using the parameter file from HPL augmented by 5 lines:

```
32 ##### This line (no. 32) is ignored (it serves as a separator). #####
33 0                                     Number of additional problem sizes for PTRANS
34 1200 10000 30000                      values of N
35 0                                     number of additional blocking sizes for PTRANS
36 40 9 8 13 13 20 16 32 64            values of NB
```

Output from each of the seven benchmarks appears in the summary output

# HPC Challenge Benchmark Output

## DGEMM output

```
DGEMM_N=288  
StarDGEMM_Gflops=2.44343  
SingleDGEMM_Gflops=2.45875
```

## PTRANS output

```
PTRANS_GBs=2.17378  
PTRANS_time=0.000628948  
PTRANS_residual=0  
PTRANS_n=500  
PTRANS_nb=80  
PTRANS_nprow=2  
PTRANS_npcol=2
```

## FFT output

```
FFT_N=32768  
StarFFT_Gflops=0.594992  
SingleFFT_Gflops=0.613019  
MPIFFT_N=262144  
MPIFFT_Gflops=6.17472  
MPIFFT_maxErr=1.28804e-15  
MPIFFT_Procs=16
```

## STREAM output

```
STREAM_VecorSize=83333  
STREAM_Threads=1  
StarSTREAM_Copy=5.14952  
StarSTREAM_Scale=5.27086  
StarSTREAM_Add=7.09093  
StarSTREAM_Triad=5.0111  
SingleSTREAM_Copy=5.33624  
SingleSTREAM_Scale=5.53154  
SingleSTREAM_Add=7.25028  
SingleSTREAM_Triad=6.75953
```

## RandomAccess output

```
MPIRandomAccess_GUPs=0.144392  
StarRandomAccess_LCG_GUPs=0.11601  
SingleRandomAccess_LCG_GUPs=0.118885  
StarRandomAccess_GUPs=0.0829133  
SingleRandomAccess_GUPs=0.083817
```

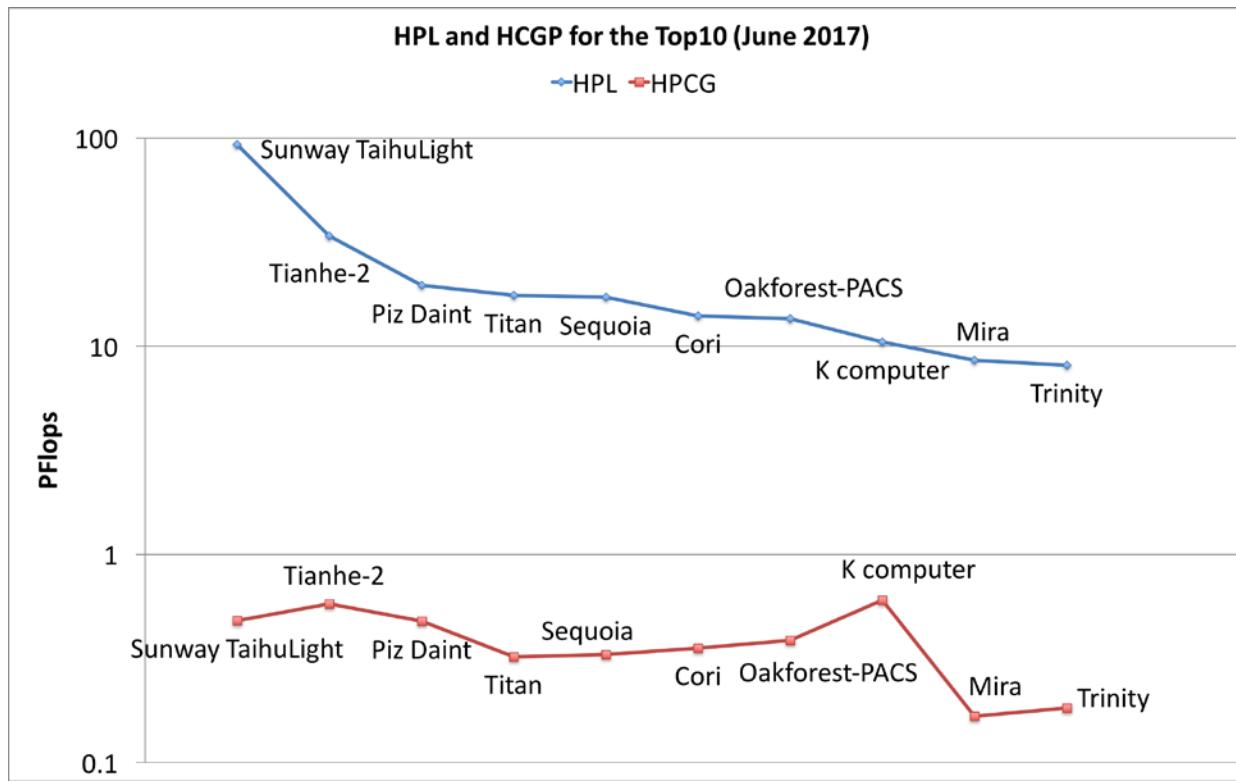
## b\_eff output

```
MaxPingPongLatency_usec=0.55631  
RandomlyOrderedRingLatency_usec=0.768096  
MinPingPongBandwidth_GBytes=4.28756  
NaturallyOrderedRingBandwidth_GBytes=0.533907  
RandomlyOrderedRingBandwidth_GBytes=0.576042  
MinPingPongLatency_usec=0.238419  
AvgPingPongLatency_usec=0.390631  
MaxPingPongBandwidth_GBytes=9.36751  
AvgPingPongBandwidth_GBytes=6.48206  
NaturallyOrderedRingLatency_usec=0.751019
```

# HPCG Benchmark

- Developed by Jack Dongarra (HPL), Michael Heroux, and Piotr Luszczek (HPL)
- Aims to complement HPL in exploring memory and data access patterns in application workloads that are not well represented by HPL
- The workload in HPCG centers on a sparse system of linear equations , i.e. solve  $Ax = b$  where A is dominated by zero entries
- Workload arises from the discretization of a three dimensional Laplacian partial differential equation with a 27 point stencil.
- For problem size n, HPCG performs  $O(n)$  floating point operations while also requiring  $O(n)$  memory accesses.
- HPCG, like HPL, reports Flops

# HPCG Benchmark for Top10 compared to HPL



***HPCG only achieves a small fraction of the HPL Flops in to Top10***

# Running HPCG

Simple parameter file:

line 3 specifies the local process cubic grid size

line 4 gives the runtime in seconds

```
1 HPCG benchmark input file
2 Sandia National Laboratories; University of Tennessee, Knoxville
3 104 104 104
4 60
```

The output includes the global problem size along with V & V'

```
HPCG-Benchmark version: 3.0
Release date: November 11, 2015
Machine Summary:
  Distributed Processes: 16
  Threads per processes: 1
Global Problem Dimensions:
  Global nx: 416
  Global ny: 208
  Global nz: 208
Processor Dimensions:
  npx: 4
  npy: 2
  npz: 2
Local Domain Dimensions:
  nx: 104
  ny: 104
  nz: 104
#####
# V&V Testing Summary #####
Spectral Convergence Tests:
  Result: PASSED
  Unpreconditioned:
    Maximum iteration count: 11
    Expected iteration count: 12
  Preconditioned:
    Maximum iteration count: 2
    Expected iteration count: 2
Departure from Symmetry |x'Ay-y'Ax|/(2*||x||*||A||*||y||)/eps
  Result: PASSED
  Departure for SpMV: 3.15835e-08
  Departure for MG: 4.00058e-09
GFLOP/s Summary:
  Raw DDOT: 4.48213
  Raw WAXPBY: 7.70723
  Raw SpMV: 7.3242
  Raw MG: 7.07338
  Raw Total: 7.05082
  Total with convergence overhead: 7.05082
Final Summary :
  HPCG result is VALID with a GFLOP/s rating of: 6.88674
```

And a final performance summary for each kernel

# NAS Parallel Benchmarks (NPB)

The NAS parallel benchmarks (NPB) are a series of small self-contained programs that encapsulate the performance attributes of a large computational fluid dynamics application

NPB	Approximate Line Count	Parallelism		Language	
		MPI	OpenMP	Fortran	C
IS – Integer Sort	1150	X	X		X
EP – Embarrassingly Parallel	400	X	X	X	
CG – Conjugate Gradient	1900	X	X	X	
MG – Multi-grid	2600	X	X	X	
FT – discrete 3D Fast Fourier Transform	2200	X	X	X	
BT – Block Tri-diagonal solver	9200	X	X	X	
SP – Scalar Penta-diagonal solver	5000	X	X	X	
LU – Lower Upper Gauss Seidel solver	6000	X	X	X	

# NAS Parallel Benchmarks (NPB)

- NPB originates from the NASA Ames research center in 1991
- The first version of the benchmark consisted of eight problems that were specified entirely in a “pencil and paper” fashion. That is, there was no reference implementation as in other benchmarks and the benchmark was specified algorithmically.
- In 1995, the second version of NPB was announced where reference versions based on MPI and Fortran77 would be distributed.
- Subsequently, a third version of NPB was released which included a number of additions to the original eight problems as some additional parallel programming API’s beyond MPI such as OpenMP, High Performance Fortran, and Java.

# NAS Parallel Benchmarks (NPB): the eight original problems

- IS: large integer sort for testing both integer computation speed and network performance
- EP: embarrassingly parallel random number generation for integral evaluation
- CG: a conjugate gradient approximation to compute the smallest eigenvalue of a sparse symmetric matrix
- MG: a multigrid solver for computing a 3-D potential
- FFT: a time integrator of a 3-D partial differential equation using the Fast Fourier Transform
- BT: a block tri-diagonal solver with a  $5 \times 5$  block size
- SP: a penta-diagonal solver
- LU: An LU solver for coupled parabolic/elliptic partial differential equations.

# Running NPB

Output from NPB IS on 4 processes

- The number of processes on which the benchmark runs is specified at compile time.
- The problem class is specified at compile time. One of 7 options: S,W,A,B,C,D, or E. W indicates a problem for a 1990's era workstation; A, B, C indicate standard problem sizes increasing by a factor of 4 with each letter; D and E indicate large test problems increasing by a factor of 16 by each letter.

NAS Parallel Benchmarks 3.3 -- IS Benchmark

Size: 65536 (class S)  
Iterations: 10  
Number of processes: 4

#### IS Benchmark Completed

Class	=	S
Size	=	65536
Iterations	=	10
Time in seconds	=	0.00
Total processes	=	4
Compiled procs	=	4
Mop/s total	=	274.91
Mop/s/process	=	68.73
Operation type	=	keys ranked
Verification	=	SUCCESSFUL
Version	=	3.3.1
Compile date	=	16 Aug 2016

#### Compile options:

MPIICC	=	mpicc
CLINK	=	\$(MPIICC)
CMPI_LIB	=	-L/usr/local/lib -lmpi
CMPI_INC	=	-I/usr/local/include
CFLAGS	=	-O
CLINKFLAGS	=	-O

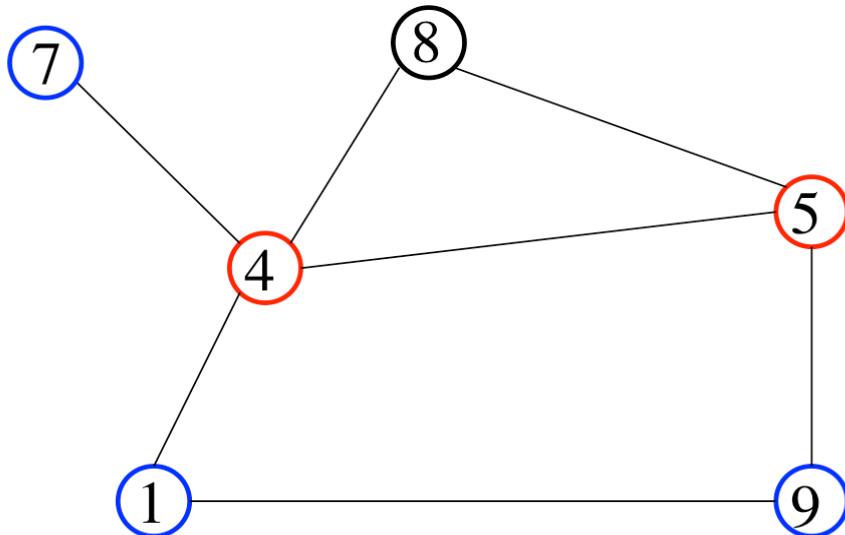
Please send feedbacks and/or the results of this run to:

NPB Development Team  
npb@nas.nasa.gov

# Graph500 Benchmark

- The graph500 benchmark was announced in 2010 and is intended to represent data intensive workloads rather than floating point intensive computations as in HPL.
- Supported by an international steering committee of over 50 members and led by Richard Murphy (Micron)
- The graph500 benchmark targets three key problems in the context of data intensive applications: concurrent search, the single source shortest path, and the maximal independent set.
- At present, only the concurrent search problem has been specified as graph500 benchmark 1 and is sometimes also referred to as the graph500 benchmark.
- The graph500 search benchmark implements the breadth-first search algorithm on a large graph.

# Graph500 Benchmark: Breadth-first search



Starting at root 8: 8, 4, 5, 1, 7, 9

Example of the breadth-first search traversal of this graph data structure starting at vertex 8. The starting vertex is also called the root. The adjacent vertices to the root are 4 and 5, colored red. The adjacent vertices to those are 1, 7, and 9, colored blue. Lines connecting the vertices are called edges

# Graph500 Benchmark 1: Search

- The benchmark starts with a root and finds all reachable vertices from that root
- 64 unique roots are checked.
- There is only one kind of edge and there are no weights between vertices.
- The output performance metric is traversed edges/second or TEPS.
- The resulting search tree is validated to ensure it is the correct tree given the root.
- The graph construction and the graph search are both timed in the graph500 search benchmark.
- The reference implementation may be downloaded from [www.graph500.org](http://www.graph500.org).
- No external libraries or dependencies are needed.

# Graph500 Benchmark 1: Search

The graph500 benchmark requires at least one input to run and can take a second input. The usage for the benchmark shown

```
Usage: ./graph500_mpi_simple SCALE edgefactor
      SCALE = log_2(# vertices) [integer, required]
      edgefactor = (# edges) / (# vertices) = .5 * (average vertex degree) [integer, defaults to 16]
      (Random number seed and Kronecker initiator are in main.c)
```

The first input supplies the code with the number of vertices:

$$N_{vertices} = 2^{scale}$$

The number of edges is given by the product of the number of vertices and the edgefactor (the second benchmark argument)

*Problem sizes in the graph500 search benchmark are classified into six categories: toy, mini, small, medium, large, and huge. These are also referred to as levels 10-15 with level 10 toy and level 15 huge.*

$$N_{edges} = \text{edgefactor} * N_{vertices}$$

Level	Scale	Size	Vertices (Billions)	Terabytes
10	26	Toy	0.1	0.02
11	29	Mini	0.5	0.14
12	32	Small	4.3	1.1
13	36	Medium	68.7	17.6
14	39	Large	549.8	141
15	42	Huge	4398.0	1,126

# Graph500 Benchmark 1: Search

Problem sizes in the graph500 search benchmark are classified into six categories: toy, mini, small, medium, large, and huge. These are also referred to as levels 10-15 with level 10 toy and level 15 huge.

Level	Scale	Size	Vertices (Billions)	Terabytes
10	26	Toy	0.1	0.02
11	29	Mini	0.5	0.14
12	32	Small	4.3	1.1
13	36	Medium	68.7	17.6
14	39	Large	549.8	141
15	42	Huge	4398.0	1,126

# Running the graph500 benchmark

First, the timing output for the graph generation and construction will be printed to screen

```
graph_generation:          0.115093 s
construction_time:        0.224907 s
```

Afterwards, the timing for the breadth-first search kernel will print to screen for each of the 64 roots followed by a validation phase, shown partially here

```
Running BFS 0
Time for BFS 0 is 0.007095
Validating BFS 0
Validate time for BFS 0 is 1.805835
TEPS for BFS 0 is 1.15464e+06
Running BFS 1
Time for BFS 1 is 0.000358
Validating BFS 1
Validate time for BFS 1 is 2.007691
TEPS for BFS 1 is 2.28912e+07
Running BFS 2
Time for BFS 2 is 0.000500
Validating BFS 2
Validate time for BFS 2 is 1.967331
TEPS for BFS 2 is 1.63852e+07
```

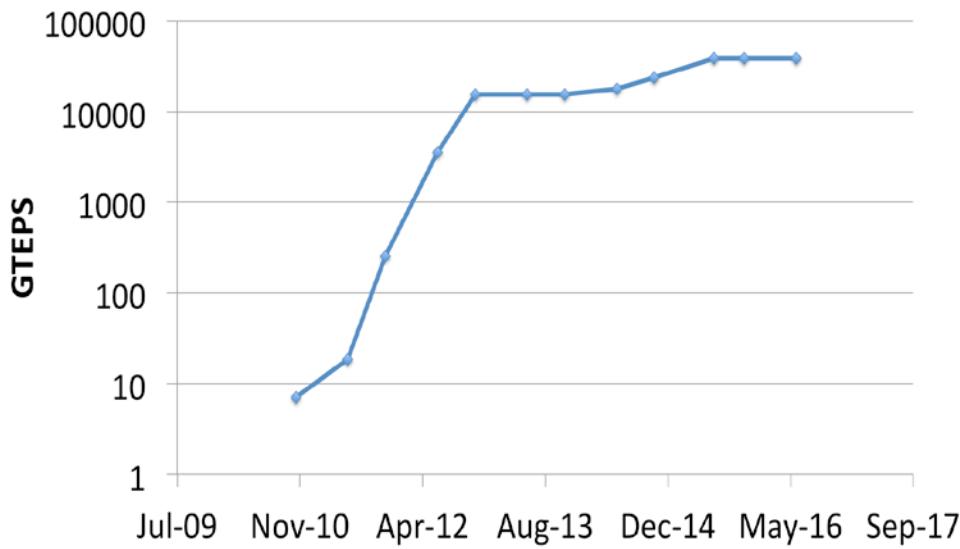
# Running the graph500 benchmark

- At the end of the graph traversal for each of the 64 roots, the final statistics for the graph500 search benchmark will print to screen
- The graph500 search benchmark does not output Flops but rather TEPS

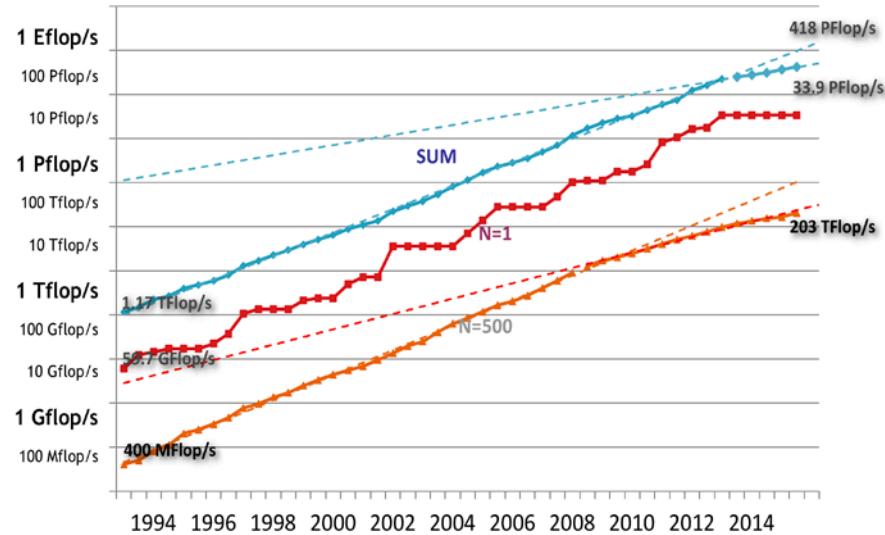
SCALE:	9
edgefactor:	16
NBFS:	64
graph_generation:	0.115093
num_mpi_processes:	16
construction_time:	0.224907
min_time:	0.000169039
firstquartile_time:	0.000205517
median_time:	0.000227094
thirdquartile_time:	0.000342607
max_time:	0.0959749
mean_time:	0.00589841
stddev_time:	0.019746
min_nedge:	8192
firstquartile_nedge:	8192
median_nedge:	8192
thirdquartile_nedge:	8192
max_nedge:	8192
mean_nedge:	8192
stddev_nedge:	0
min_TEPS:	85355.6
firstquartile_TEPS:	2.39107e+07
median_TEPS:	3.60732e+07
thirdquartile_TEPS:	3.98605e+07
max_TEPS:	4.84623e+07
harmonic_mean_TEPS:	1.38885e+06
harmonic_stddev_TEPS:	585773
min_validate:	1.72823
firstquartile_validate:	1.86437
median_validate:	1.91839
thirdquartile_validate:	2.00359
max_validate:	2.11599
mean_validate:	1.92975
stddev_validate:	0.0889681

# Graph500: Observation 1

Graph500 Search Benchmark: Best performance over time

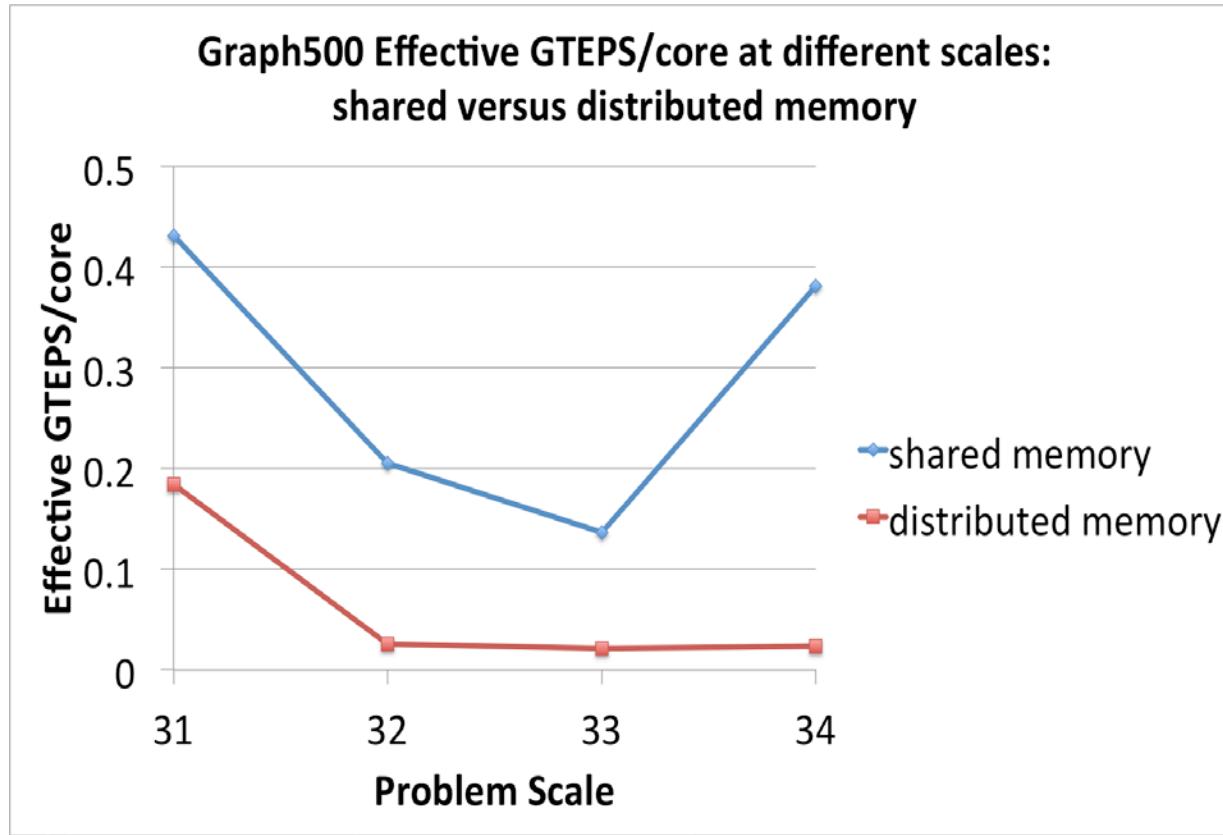


Performance Development



While the HPL benchmark continues to show exponential improvement on newer supercomputers, the graph500 search benchmark performance has gone flat.

# Graph500: Observation 2



The effective GTEPS/core is much lower for distributed memory architectures than for shared memory.

# Mini-applications

- Many argue that the HPC benchmarks are too simple in order to properly assess a supercomputer's performance with respect to a dynamic application
- To complement HPC benchmarking efforts and better capture real application behavior, many in the HPC community have turned to using mini-applications.
- Like the name implies, mini-applications are smaller versions of real applications
- They originate from a large number of scientific disciplines and are generally much longer than HPC benchmarks
- They do not generally output any standardized metric like Flops, GUPS, TEPS, or DOFS but do provide time to solution for various kernels as well as strong and weak scaling information

# Mini-applications

Some standard mini-applications from the Mantevo suite

Mini-Application	Approximate Line Count	Parallelism			Language		
		MPI	OpenMP	Other	Fortran	C	C++
MiniAMR	9400	X				X	
MiniFE	14200	X	X	CUDA,Cilk			X
MiniGhost	12770	X	X	OpenACC	X		
MiniMD	6500	X	X	OpenCL, OpenACC			X
CloverLeaf	9300	X	X	OpenACC, CUDA	X	X	
TeaLeaf	6500	X	X	OpenCL	X		

# Mini-applications

- Mini-applications fulfill several roles that are difficult for standard HPC benchmarks.
- Mini-applications enable large application developers to interact with a broader software engineering community by producing simplified, smaller, open source versions of their application
- Mini-applications also serve an important role in testing emerging programming models.
- Mini-applications are well suited for performing scaling studies, especially in the context of emerging hardware architectures.
- Mini-applications are sufficiently complex yet small enough to explore the parameter and interaction space of memory, network, accelerators, and processor elements.
- Mini-applications are frequently used in supercomputer procurement decisions.

# Resource Management

# Need for Resource Management

- Protecting significant infrastructure investment
  - Supercomputing hardware
  - Hosting data center
  - Power delivery (cabling, UPSs, generators, flywheels)
  - Cooling infrastructure (HVAC units, heat exchangers, pipes for water cooling, etc.)
- Cost of ownership
  - Electricity (about \$1M/1MW per year)
  - Support contracts
  - Maintenance crew (system administrators, technicians)
- Getting the most for your \$
  - Efficient workload scheduling
  - Maximizing hardware resource utilization
  - Compute job prioritization

# Types of Managed Resources

- Compute nodes
- Processing cores
- Interconnect
- Permanent storage and I/O
- Accelerators

# Compute Nodes

- Each node provides a fixed compute and storage footprint
- Constitutes a fundamental resource allocation unit for a compute job
- Increasing the number of nodes enables straightforward application scaling
- Resources may vary depending on node type
  - CPU type and clock frequency
  - Main memory capacity
  - Interconnect bandwidth and latency
  - Optionally: secondary storage speed and capacity

# Processing Cores

- Provide local parallelism:
  - Multiple cores per processor
  - Possibly multiple processors (sockets) per node
- Sharing of processing elements between different applications:
  - Depends on application type and usage context
  - Single compute core is a typical allocation unit
  - Shared mode enables better resource utilization
    - Parameter space sweeps
    - Large number of unrelated small jobs
  - Exclusive mode allocates full node to a single application
    - Benchmarking (minimizes intrusions from unrelated jobs)
    - Applications that demand large memory capacity
    - Applications generating large number of I/O requests to local storage
    - Bandwidth-intensive distributed jobs
    - Affinity to other hardware components of the node (e.g., PCI-Express)

# Interconnect

- Typically one type per system
  - 10G Ethernet
  - InfiniBand
  - 1G Ethernet (GigE)
- Installations with multiple networks possible
  - To permit experiments with different bandwidth and latency profiles
  - To accommodate bandwidth-insensitive applications along with network “hogs”
  - To enable high-performance mechanisms available in certain implementations (such as RDMA)
  - To utilize channel bonding for increased communication bandwidth

# Permanent Storage

- Typically implemented as a shared file system
  - Exported system-wide, thus accessible from every node
  - High capacity to accommodate all users
  - Optimized for performance and reliability (e.g., RAID)
- Multiple file systems may be provided for different usage scenarios
  - Users' home directories, often quota-limited
  - Scratch space for larger data volumes, but with limited retention (e.g. automatic deletion after one week of last access)
  - Archive storage with very large capacity, long term retention, but potentially limited access bandwidth
  - Local file systems
- Localized storage may provide better aggregate data bandwidth
  - Example 1: disk drives in each node
  - Example 2: burst buffers in Cray machines (SSD pools shared by groups of nodes)
  - But: limited data access options (the user must log into the same machine or group of nodes where dataset was saved)

# Accelerators

- Special purpose processors
  - Heterogeneous (different hardware class to main CPU)
  - May be application specific
  - Expose large number of fine-grain processing elements
  - Consume less energy and time to accomplish certain tasks compared to the host processor
  - Use different compilers and libraries to generate executables
  - Knowledge of internal organization usually a must to develop efficient applications
  - Often require explicit code and data offload to processing unit
- Commonly used variants
  - General Purpose Graphics Processing Unit (GPGPU)
  - Intel Xeon Phi
  - Field Programmable Gate Array (FPGA)
- System hardware may not be uniformly populated with accelerators
  - Some nodes may be equipped with several accelerators while the others contain none
  - Resource managers should support allocation of accelerated nodes when required by application

# Compute Job

- A self-contained work unit that may need specific input data and produces a result in the course of its execution (output)
  - Input may be typed by the user or a file
  - Output could be a file, a printout on the console or graphics displayed on screen
- Primary modes of job processing
  - Interactive: requires interaction with user
  - Batch: resource requirements, job description, and inputs are fully specified at job submission time
- Batch processing accounts for most of machine's computational throughput
- May be subdivided into smaller *steps* or *tasks*
  - Data staging
  - Data pre- and post-processing
  - Visualization
  - Processing application pipelines

# Job Queue

- Defines order in which jobs are executed on the machine
  - Typically FIFO (“first-in, first-out”)
  - Job schedulers may deviate from the prearranged order to improve utilization/throughput or due to operator’s override
- Groups jobs targeting the same set of resources
  - Most systems have multiple job queues, for example
    - Production
      - Job size
      - Job duration
    - Debug
    - Interactive
    - Managing specific resources, such as large memory nodes or accelerators

# Job Scheduling

- Critical to achieving good utilization of the machine
- Must accommodate potentially hundreds or thousands of waiting jobs and a significant number of concurrently executing jobs
- Parameters affecting job scheduling
  - Resource availability
  - Job priority
  - Resources allocated to the user
  - Maximum number of jobs
  - Requested execution time
  - Elapsed execution time
  - Job dependencies and prerequisites
  - Occurrence of specified events
  - Operator availability
  - Software license availability

# Common Resource Managers

- Simple Linux Utility for Resource Management (Slurm)
- Portable Batch System (PBS)
- OpenLava based on the Load Sharing Facility (LFS)
- Moab Cluster Suite
- Tivoli Workload Scheduler LoadLeveler
- Univa Grid Engine
- HTCondor
- OAR
- Hadoop Yet Another Resource Negotiator (YARN)

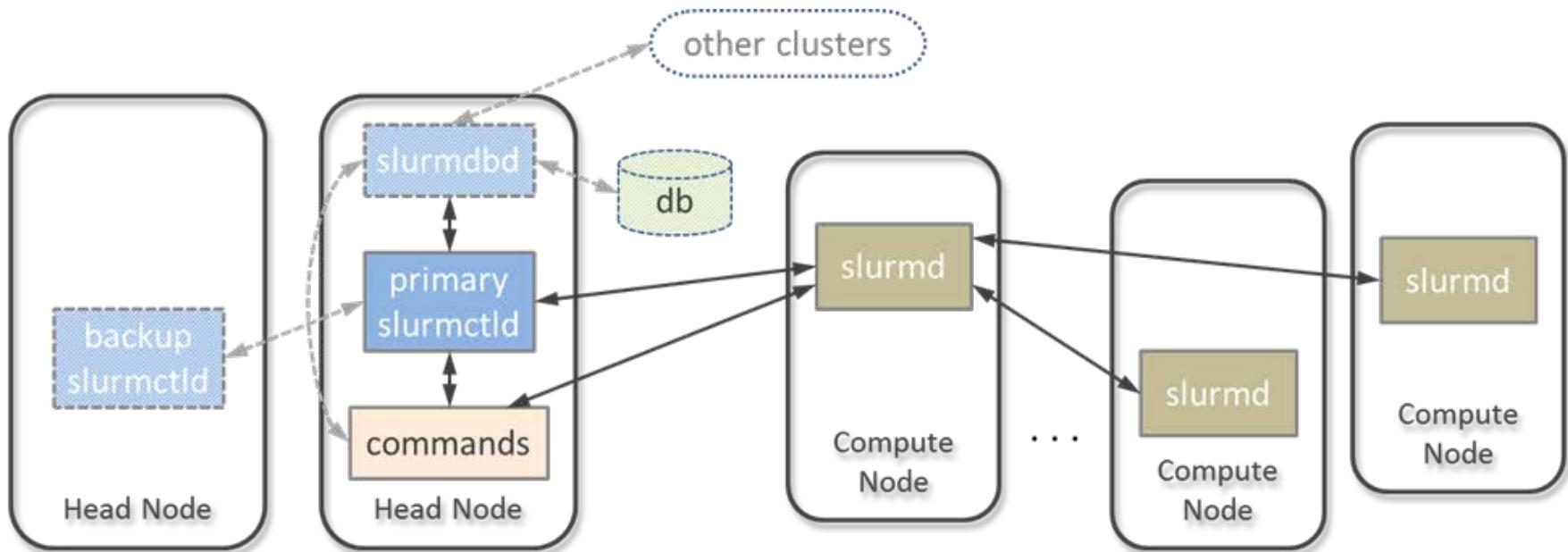
# Slurm Introduction

- One of the most popular open-source managers
- Originated in early 2000s at Lawrence Livermore National Laboratory (Morris Jette)
- Currently close to 200 contributors (such as SchedMD LLC, HP, Bull, Cray, ORNL, LANL, Intel, Nvidia, and others)
- Utilized in about 60% of machines in the TOP500 list (mid-2014)

# Slurm Features

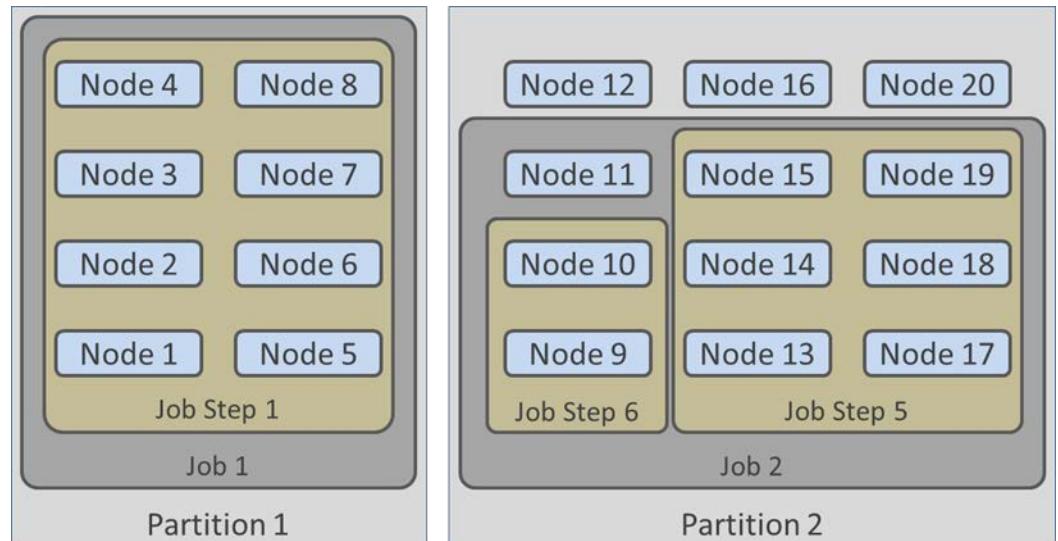
- Complex configuration options
  - Different interconnect types
  - Scheduling algorithms
  - MPI implementations
  - Job accounting
- Scalable to over 10 million cores (TaihuLight)
- Manages up to a thousand job submissions and 500 job executions per second
- Multiple strategies for power optimization
- Increased reliability through multiple daemons
- Network topology aware
- Decisions influenced by node architecture (NUMA, core distribution, thread affinities)
- Supports expanding and shrinking job's resource footprint during its execution
- A number of scheduling algorithms with a broad range of control parameters
- Accelerator support
- Extensible via plugins in C and Lua
- Optional support for job profile database

# Slurm Architecture



# Workload Organization

- Partitions (queues)
  - Group execution resources
  - Uniquely named
  - May overlap
- Job steps may utilize all or a fraction of nodes allocated to a parent job
- Job arrays are a means to manage a large number of similar jobs (e.g., for throughput computing)



# Slurm Scheduling

- Event-triggered scheduling as default
  - Considers only a limited number of jobs at the head of the queue
  - Fast and low overhead
- Backfill scheduler
  - Starts lower priority jobs if that will not affect the expected start time of higher priority jobs
  - Fair treatment of lower priority jobs that require large amount of resources
  - Improves overall utilization
- Gang scheduling
  - For multiple similar jobs sharing the same set of resources
- Preemption
  - Stopping lower priority jobs to launch higher priority jobs
- Generic Resources (GRES) driven scheduling
  - Based on hardware device availability (e.g., accelerators)
- Elastic computing
  - Permits growing or shrinking of the overall resource footprint (both provided by the system or requested by the job)
- High throughput computing

# *srun*

- Starts a parallel job or job steps
- Performs resource allocation for the job if such has not been performed yet
- Syntax: srun [<options>] <executable> [<arguments>]
- Frequently used options:
  - t or --time=<[hours:]minutes:seconds>
  - p or --partition=<name>
  - N or --nodes=<min\_nodes>
  - n or --ntasks=<number\_of\_tasks>
  - c or --cpus-per\_task=<number\_of\_cpus>
  - mem=<megabytes>
  - exclusive
  - s or --oversubscribe

# *salloc*

- Performs resource allocation and runs the command specified by the user
- Accepts the same options as *srun*
- Syntax: `salloc [<options>] [<command> [<arguments>]]`

# *sbatch*

- Submits a job script for batch processing
- Exits as soon as job parameters are accepted by the controller daemon (it does not wait for job execution)
- Job output is stored in file named “slurm-%j.out”, where %j is the job number
- Syntax: `sbatch [<options>] [<script> [<arguments>]]`
- Uses the same resource options as *srun*

# Example MPI+OpenMP Job Script

```
#!/bin/bash
#SBATCH --nodes=16
#SBATCH --cpus-per-task=8
#SBATCH time=2:00:00

# The hello_world application will be started
# on 16 nodes with 8 threads per process;
# its execution time is limited to 2 hours

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
mpirun hello_world
```

# *squeue*

- Displays information about queued jobs
- May be used to examine the job's status as well as resource allocations
- Syntax: squeue [<options>]
- Frequently used options:
  - all
  - l or --long
  - M or --clusters=<list\_of\_clusters>
  - p or --partition=<name>
  - t or --states=<list\_of\_states>
  - u or --user=<list\_of\_users>
  - i or --iterate=<seconds>
- Common job states include (abbreviated form in parentheses): PENDING (PD), RUNNING (R), SUSPENDED (S), STOPPED (ST), COMPLETING (CG), CANCELLED (CA), FAILED (F), TIMEOUT (TO), PREEMPTED (PR)

# *scancel*

- Used to cancel submitted jobs or deliver signals to them
- Syntax: scancel [<options>] [<jobid>]...
- Frequently used options:
  - s or --signal=<signal\_name>
  - n or --name=<job\_name>
  - p or --partition=<partition\_name>
  - t or --state=<job\_state>
  - u or --user=<user\_id>
  - i or --interactive
- The permitted job state include PENDING, RUNNING or SUSPENDED

# *sacct*

- Retrieves job information from Slurm logs
- May be used to access status and exit code of no longer existing jobs
- Syntax: `sacct [<options>]`
- Frequently used options:
  - a or allusers
  - L or allclusters
  - l or long
  - j or job=<job\_id>
  - name=<list\_of\_job\_names>
  - s or state\_list=<list\_of\_states>
- The permitted states are same as for the *queue* command

# *sinfo*

- Displays system's partition and node information
- Syntax: `sinfo [<options>]`
- Frequently used options:
  - *queue*'s options: all, long, clusters partition, iterate have the same meaning here
  - r or --responding
  - d or --dead
  - e or --exact
  - N or --Node

# Important Environment Variables

- Accessible from within job scripts
- Permit runtime customization of job execution
- Often used variable list
  - SLURM\_NTASKS
  - SLURM\_NTASKS\_PER\_NODE
  - SLURM\_CPUS\_PER\_TASK
  - SLURM\_JOB\_NUM\_NODES (total number of nodes allocated to job)
  - SLURM\_CPUS\_ON\_NODE (number of cores on the current node)
  - SLURM\_SUBMIT\_HOST
  - SLURM\_CLUSTER\_NAME
  - SLURM\_JOB\_PARTITION
  - SLURM\_JOB\_ID
  - SLURM\_JOB\_NODELIST (list of node names allocated to the job)
  - SLURM\_TASKS\_PER\_NODE (list of tasks on each node corresponding to the host order in SLURM\_JOB\_NODELIST)
  - SLURM\_SUBMIT\_DIR

# Slurm Tips and Tricks

- Launch interactive shell:

```
srun -N2 -t60 --pty /bin/bash
```

- Enable X window forwarding:

```
srun -N1 -t20 --x11 xterm
```

- Figure out the estimated start time of queued job:

```
squeue --start -j <job_id>
```

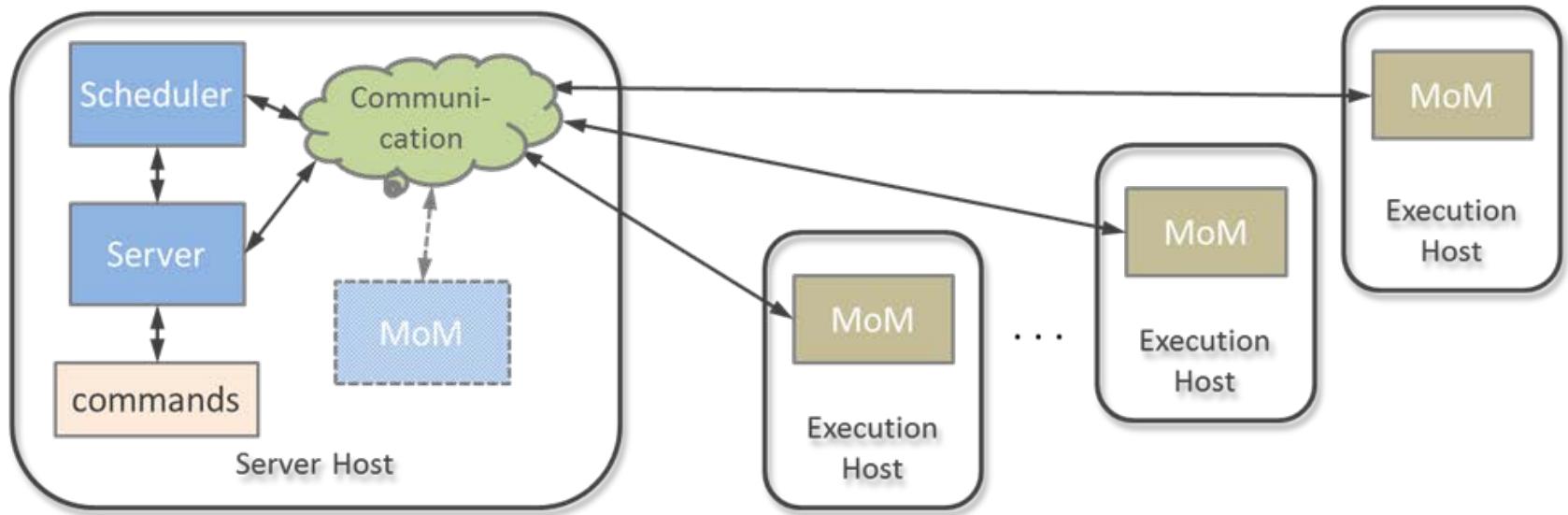
- Email notification when job terminates or fails:

```
sbatch --mail-type=END,FAIL myjobsript
```

# PBS Overview

- Started as a contract for NASA Ames developed by MRJ Technology Solutions in 1991
- Based on POSIX 1003.2d standard
- Includes contributions from NASA Ames, LLNL, NERSC
- Commercial version (PBS Pro) released by Veridian Corp. in 2000
- IP acquired by Altair Engineering (current owner)
- Open sourced in 2016
- Capable of grid workload execution using Globus toolkit
- Supports advanced reservation, ability to dynamically add and remove execution nodes, topology-aware scheduling, GPUs
- Used by many commercial and academic institutions as well as national laboratories
- Other, mostly compatible, implementations include:
  - OpenPBS (no longer maintained)
  - TORQUE (Terascale Open-source Resource and QUEue manager) developed by Adaptive Computing

# PBS Architecture



- Server daemon processes user commands and creates, monitors, and dispatches jobs
- Scheduler monitors system resources (by polling MoMs) and allocates them to jobs
- Machine Oriented Mini-servers (MoMs) are a job executors
- Servers may be replicated in larger systems

# *qsub*

- Submits a job to batch processing system
- Syntax: qsub [<options>] <script>
- Frequently used options:
  - l <resource\_list> (explained in the next slide)
  - q <queue\_name>
  - N <job\_name>
  - V (export environment variables to job's environment)
  - o <path> (capture standard output to a file)
  - e <path> (capture standard error to a file)

# Specifying Resources in PBS

- Syntax: <name>=<value>[,<name>=<value>...]
- Resource types:
  - Nodes  
nodes=<node\_count>[:ppn=<processors\_per\_node>]
  - Wall time  
walltime=[[<hours:>]<minutes>:]<seconds>
  - Memory  
mem=<integer>[<unit>]  
Where <unit> may be “B” (bytes) or “W” (words), optionally prefixed with “K” (kilo), “k” ( $\times 2^{10}$ ), “M” (mega), “m” ( $\times 2^{20}$ ), “G” (giga) or “g” ( $\times 2^{30}$ )
  - Accelerator specification is site-dependent, often:  
gpu=<integer>

# New Resource Specification Format

- Incompatible with the old style (don't mix them!)
- Uses the concept of virtual node (vnode)
- Identified by keyword "select"
- Syntax: select=[<number>:]<chunk>[+<number:>]<chunk>...]
  - <number> determines how many instances of <chunk> should be allocated
  - <chunk> is a list of <resource>=<value> assignments, separated by colons ":"
  - Common resource names:
    - ncpus (number of cores)
    - mem (physical memory per chunk)
    - mpiprocs (number of MPI processes per chunk)
    - naccelerators (number of accelerators on host)
    - accelerator\_memory (amount of required accelerator memory)
    - accelerator\_model (type of accelerator)
    - ompthreads (number of OpenMP threads to use)

# New Placement Specification

- Syntax: place=[<arrangement>][:<sharing>][:<grouping>]
- Arrangement is one of
  - free (place job on any vnode)
  - pack (puts all chunks on a single host)
  - scatter (assigns one MPI chunk to a host)
  - vscatter (one chunk per vnode)
- Sharing may be
  - excl (vnode exclusive to the current job)
  - shared (vnode shared with other jobs)
  - exclhost (entire host exclusive to the job)
- Grouping determines how chunks are grouped according to the resource
  - Syntax: <group>=<resource>
  - Some resources may be site specific

# Examples

- Allocate 24 nodes for half an hour to a job:  
`qsub -l nodes=24 -l walltime=30:00 short_job.sh`
- Submit a job to execute on two specific hosts for two hours  
`qsub -l nodes=host005+host006 -l 2:00:00 postprocess.sh`
- Allocate 16 chunks with four processors and 1GB memory each using new format:  
`qsub -l select=16:ncpus=4:mem=1gb mpi64.sh`

# *qdel*

- Deletes one or more jobs from the system
- Syntax: qdel [<options>] <job\_id> ...
- Frequently used options:
  - W force (forcible deletion)
  - x (removes all jobs, including moved and finished jobs, and their history)

# *qstat (l)*

- Displays status of jobs, queues or servers
- Job status syntax: qstat [<options>][<job\_id>]
  - a (reports running and queued jobs)
  - H (shows finished and moved jobs)
  - i (displays information about waiting, held, and queued jobs)
  - r (shows running and suspended jobs)
  - t (lists jobs, job arrays, and subjobs)
  - u <user>[,...] (restricts display to jobs owned by specific user(s))
  - f (enables long format containing job ID, attributes, submission arguments, and executable with arguments)
  - p (replaces time use value with completion percentage)

# *qstat (II)*

- Queue status syntax: qstat -Q [-f] [<queue>]
  - f (shows full status, one attribute per line)
- Server status syntax: qstat -B [<options>] [<server>]
  - f (shows full status)
  - G (shows size in gigabytes)
  - M (shows size in megawords)

# *tracejob*

- Outputs logged information about a job
- Syntax: tracejob [<options>] <job\_id>
- Frequently used options:
  - v (increases verbosity)
  - c <number> (shows <number> of most recent occurrences)
  - n <days> (accesses history no more than <days> old)
  - a (suppresses accounting information)
  - l (suppresses scheduling information)
  - m (suppresses MoM information)
  - s (suppresses server information)

# *pbsnodes*

- Examines status of system hosts
- Syntax: pbsnodes [<options>] [<host> ...]
- Frequently used options:
  - a (lists all hosts and their attributes)
  - j (displays job related information, including vnodes and their state)
  - S (shows system oriented information, including OS resources and custom hardware)
  - H <host>[,<host>...] (reports attributes with non-default values for specified hosts)
  - L (produces long format output)

# Example MPI+OpenMP PBS Job Script

```
#!/bin/bash
#PBS -l select=32:ncpus=8:ompthreads=8
#PBS -l walltime=1:30:00

# The script describes an MPI job to be run on
# 32 vnodes with 8 cores each for up to 1.5 hour.
# Since the environment is not automatically
# propagated, the script loads MPI environment
# explicitly (this part may be system-specific).

module load openmpi
mpirun mpisim simfile2.hdf5
```

# Important PBS Environment Variables

- PBS\_NODEFILE – path to file listing allocated execution vnodes
- NCPUS – number of available threads per vnode
- PBS\_TASKNUM – number of Unix process executing on this vnode
- PBS\_JOB\_ID – job identifier
- PBS\_JOBNAME – user-defined job name
- PBS\_QUEUE – name of the queue the job was submitted to
- PBS\_JOBDIR – job's execution directory
- PBS\_TMPDIR – job-specific temporary directory
- PBS\_O\_WORKDIR – job submission directory (absolute path)
- PBS\_O\_HOME – value of HOME variable in submission environment
- PBS\_O\_HOST – name of job submission host

# PBS Tips and Tricks

- Interactive execution:

```
qsub -I -l nodes=1,walltime=40:00
```

- X window forwarding:

```
qsub -X -I -l nodes=2,walltime=30:00 xterm
```

- Submitting job to the specific queue:

```
qsub -q -l nodes=4,walltime=30:00 myjob.sh
```

- Find out the estimated start of execution:

```
qstat -T <job_id>
```

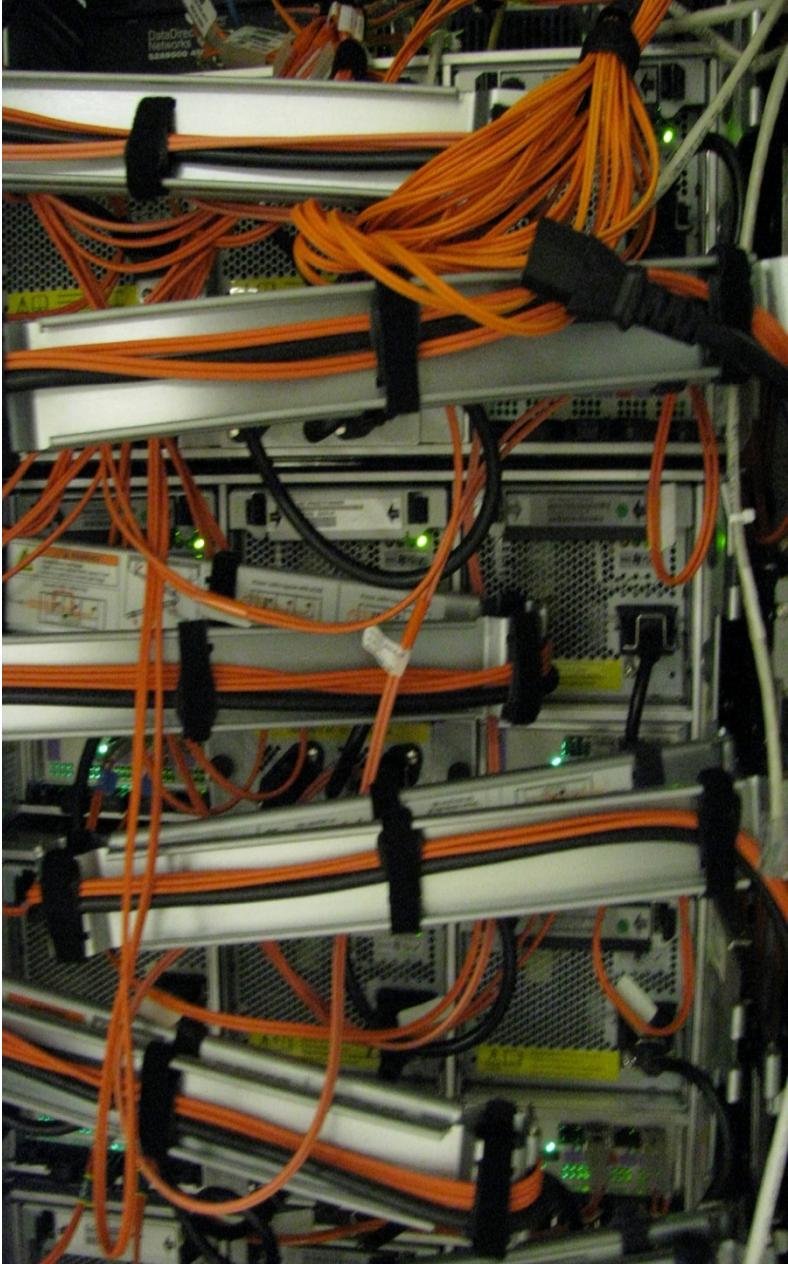
- Email notification when job aborts or terminates:

```
qsub -m ae -l nodes=4,walltime=2:00:00 job.sh
```

# Symmetric Multiprocessor Architecture

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# Opening Remarks

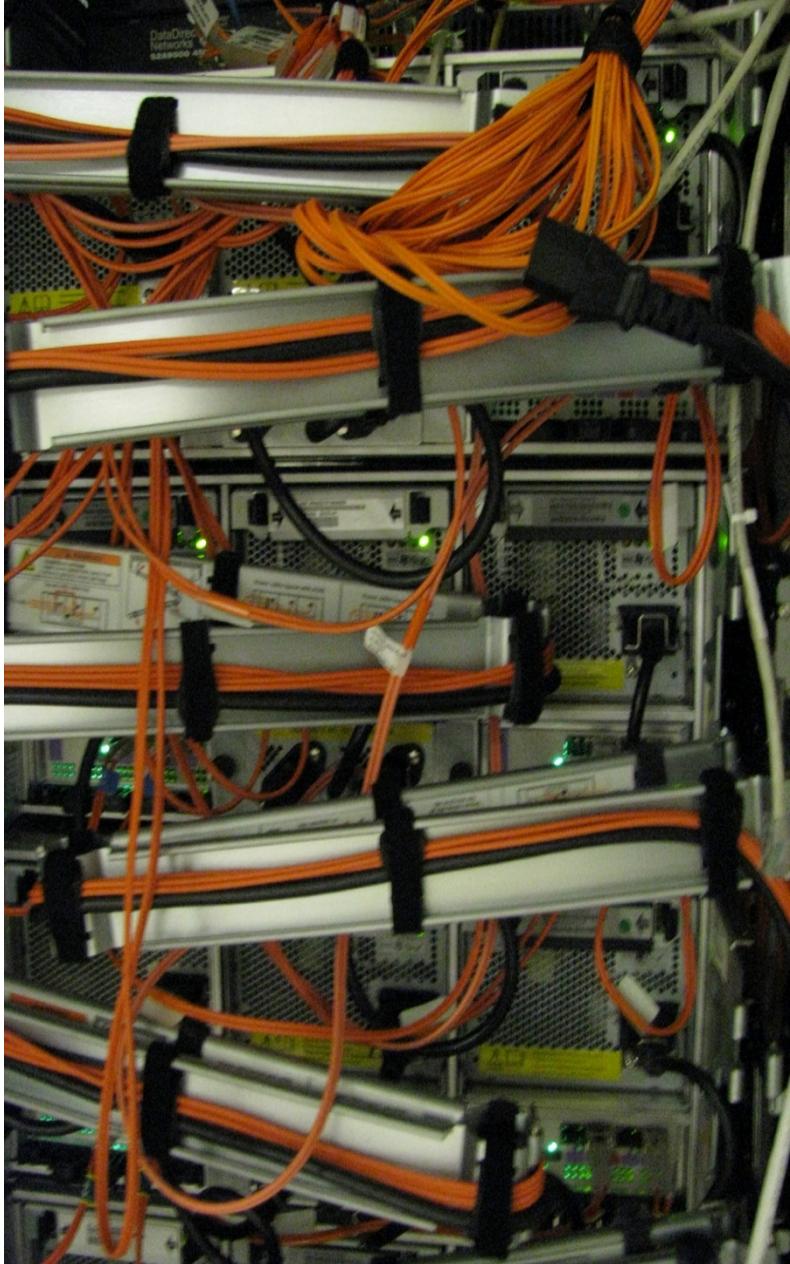
- This week is about supercomputer architecture
  - Last time: end of cooperative computing
  - Today: capability computing with modern microprocessor and multicore SMP node
- As we've seen, there is a diversity of HPC system types
- Most common systems are either SMPs or are ensembles of SMP nodes
- “SMP” stands for: “Symmetric Multi-Processor”
- System performance is strongly influenced by SMP node performance
- Understanding structure, functionality, and operation of SMP nodes will allow effective programming

# The take-away message

- Primary structure and elements that make up an SMP node
- Primary structure and elements that make up the modern multicore microprocessor component
- The factors that determine microprocessor delivered performance
- The factors that determine overall SMP sustained performance
- Amdahl's law and how to use it

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# SMP Context

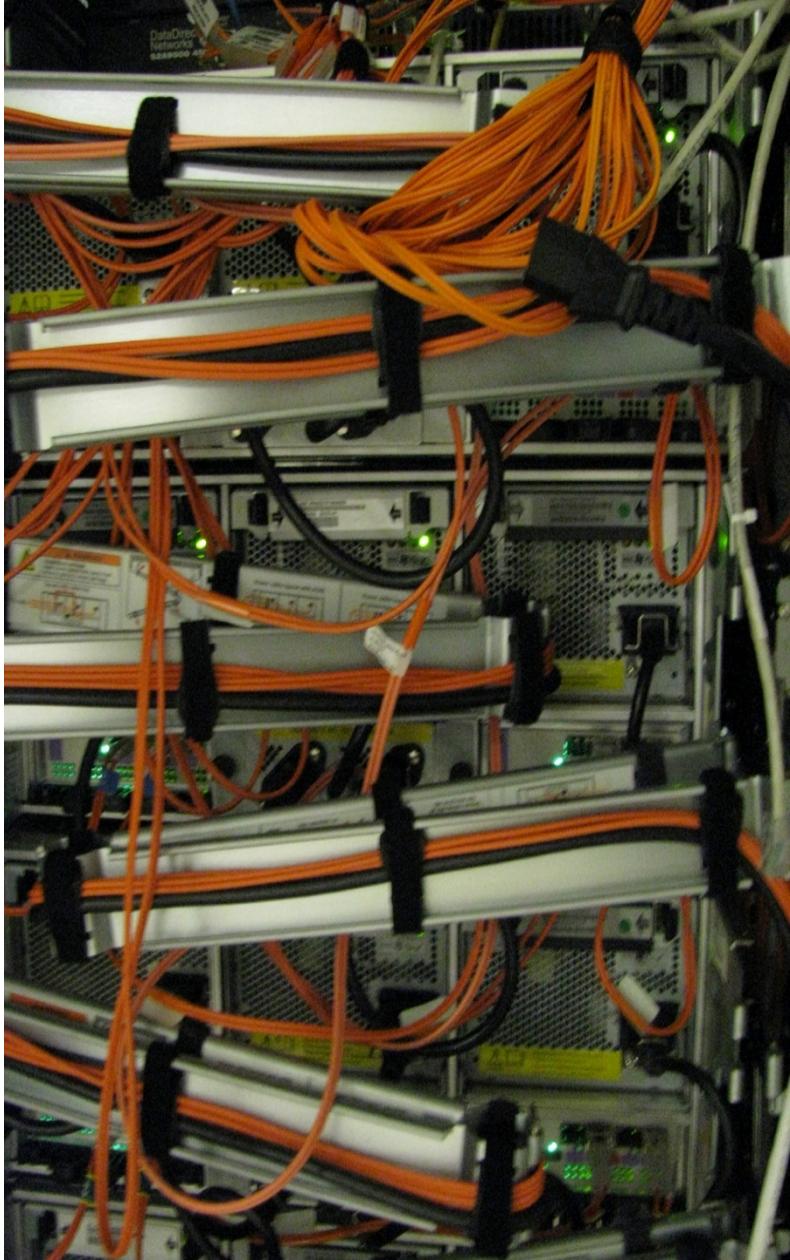
- A standalone system
  - Incorporates everything needed for
    - Processors
    - Memory
    - External I/O channels
    - Local disk storage
    - User interface
  - Enterprise server and institutional computing market
    - Exploits economy of scale to enhance performance to cost
    - Substantial performance
  - Target for ISVs (Independent Software Vendors)
- Shared memory multiple thread programming platform
  - Easier to program than distributed memory machines
  - Enough parallelism to fully employ system threads (processor cores)
- Building block for ensemble supercomputers
  - Commodity clusters
  - MPPs

**Table 6.1 Some Examples of SMPs and Their Characteristics**

Vendor and Name	Processor	Number of Cores	Cores per Central Processing Unit	Memory Capacity	PCIe Slots	Storage Slots
IBM S822LC	IBM POWER8 2.92 GHz	20	10	256 GB	2 × 16-lane Gen.3 3 × 8-lane Gen.3	12 LFF
HPE rx2800 i6	Intel Itanium 9760 2.66 GHz	16	8	384 GB	3 × 16-lane Gen.2 2 × 8-lane Gen.2	8 SFF
Dell PowerEdge R930	Intel E7-8870v4 2.1 GHz	80	20	12 TB	10 Gen.3	24 SFF 8 NVMe
Oracle SPARC T7-4	SPARC M7 4.13 GHz	128	32	4 TB	8 × 16-lane Gen.3 8 × 8-lane Gen.3	8 SFF
HPE ProLiant DL385p Gen8	AMD Opteron 6373 2.3 GHz	32	16	384 GB	3 × 16-lane Gen.2 3 × 8-lane Gen.2	8 SFF

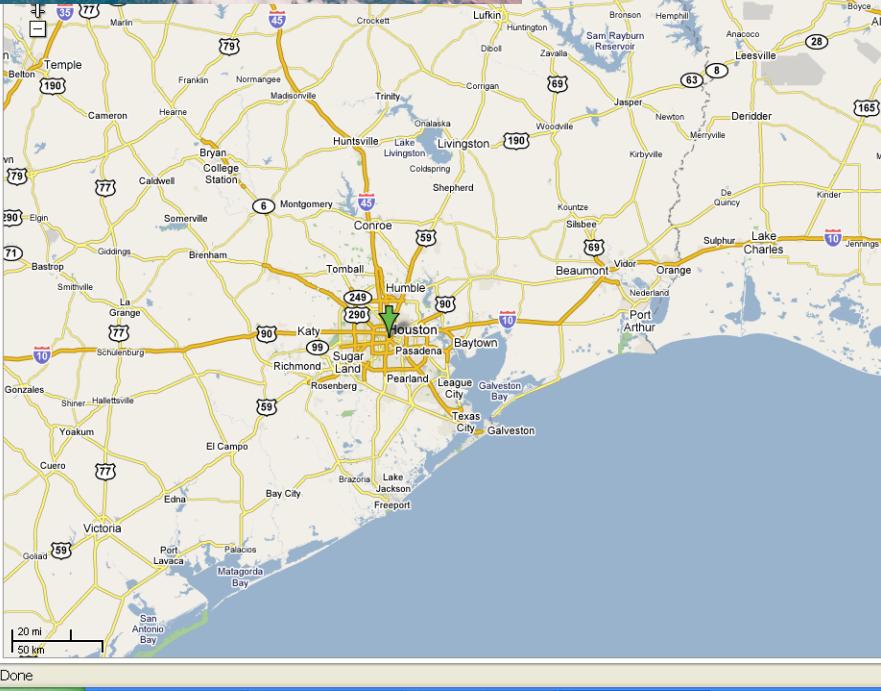
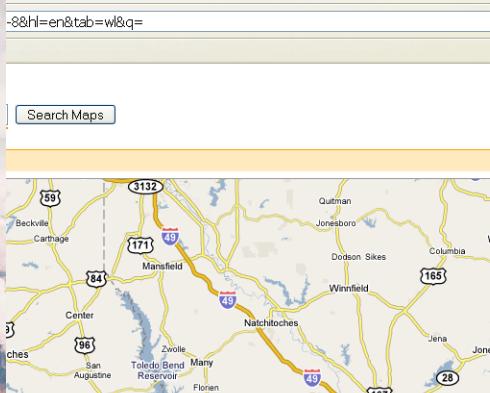
# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# Performance: Amdahl's Law

 1200 Louisiana St. Houston TX - Google Maps - Mozilla Firefox



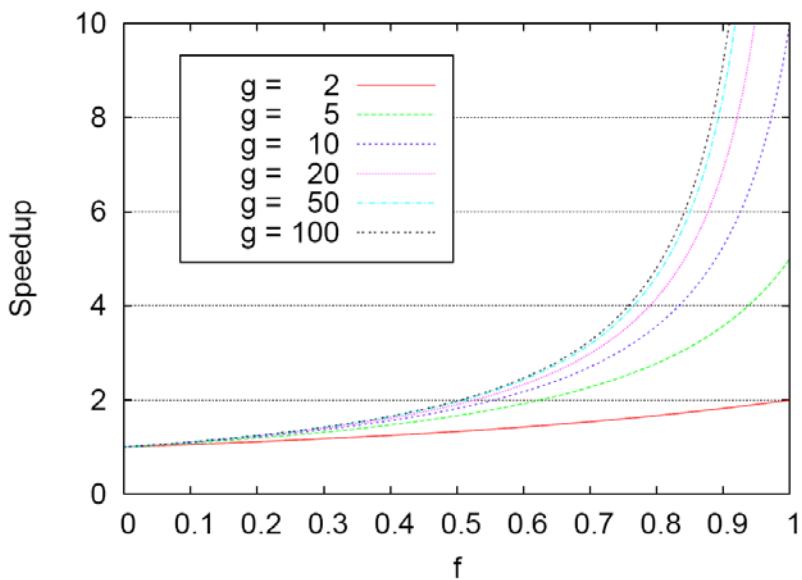
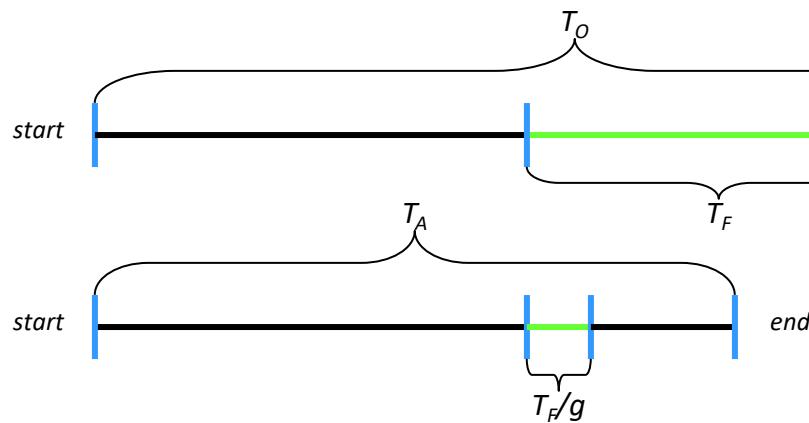
## Baton Rouge to Houston

- *from my house on East Lakeshore Dr.*
  - *to downtown Hyatt Regency*
  - *distance of 271*
  - *in air flight time: 1 hour*
  - *door to door time to drive: 4.5 hours*
  - *cruise speed of Boeing 737: 600 mph*
  - *cruise speed of BMW 528: 60 mph*

# 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

$$S = T_o / T_A$$

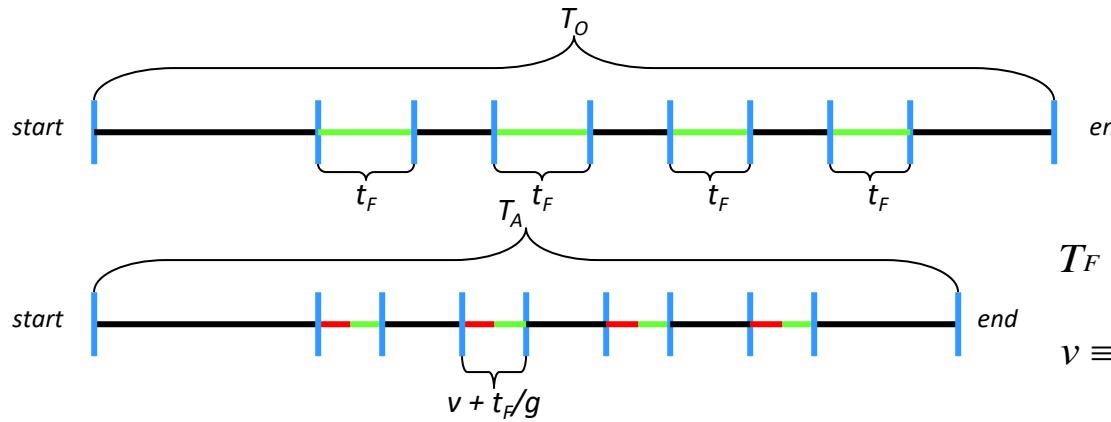
$$f = T_F / T_o$$

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

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

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

# Amdahl's Law with Overhead



$$T_F = \sum_i^n t_{Fi}$$

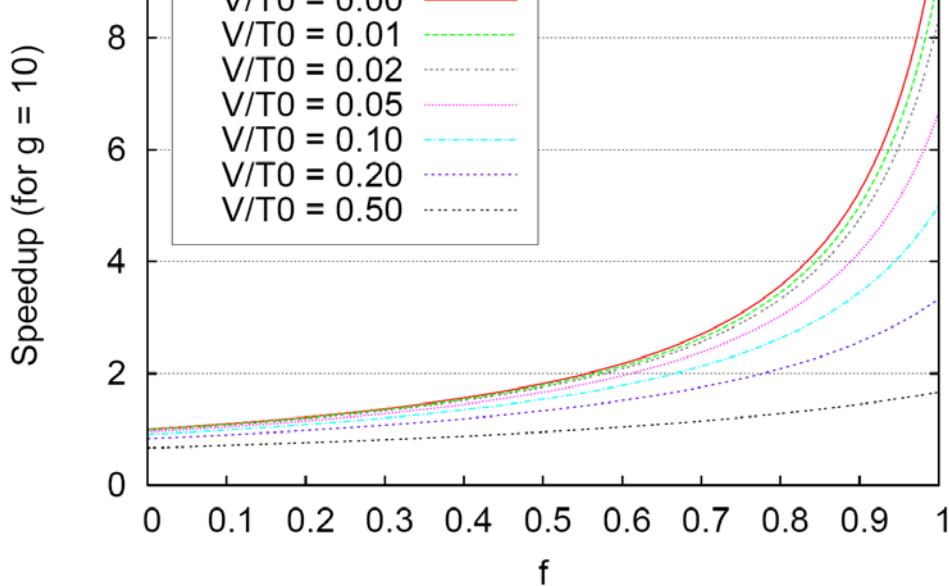
$v$   $\equiv$  overhead of accelerated work segment

$$V \equiv \text{total overhead for accelerated work} = \sum_i^n v_i$$

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

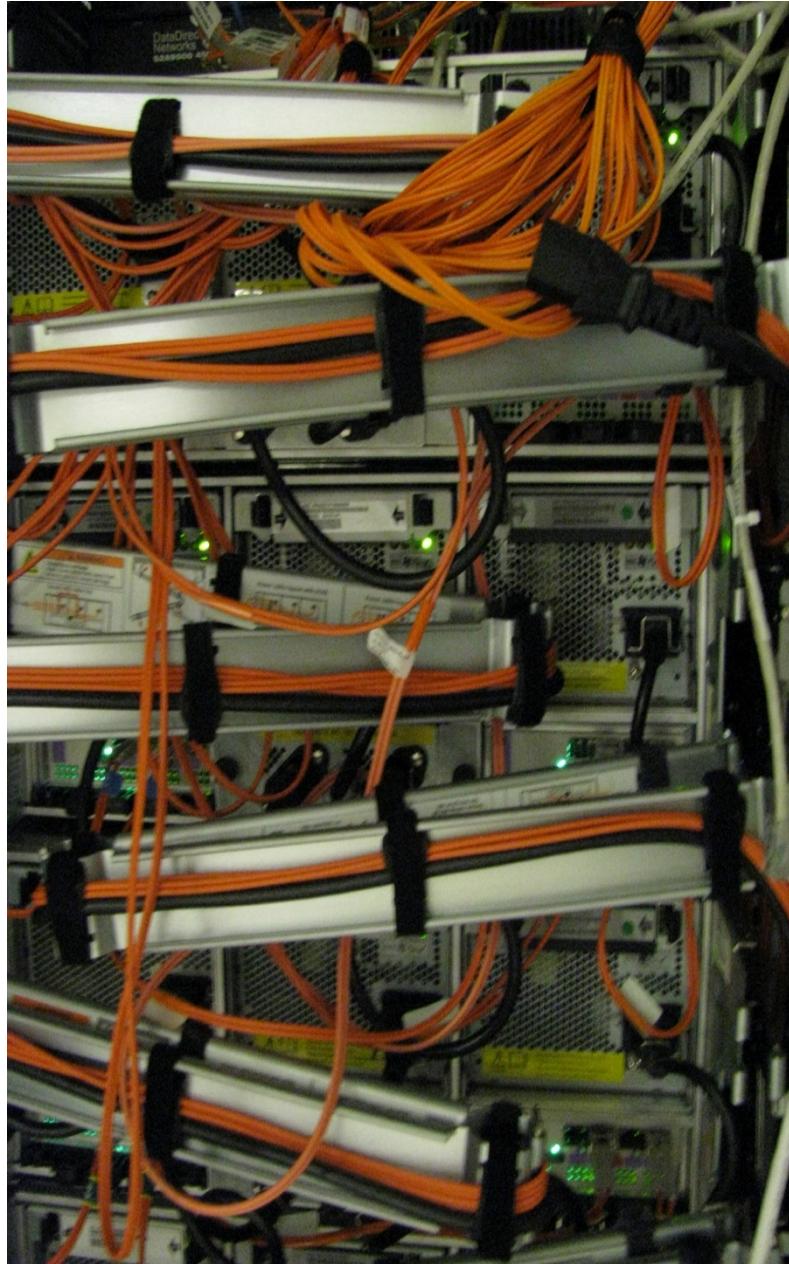
$$S = \frac{T_O}{T_A} = \frac{T_O}{(1-f) \times T_O + \frac{f}{g} \times T_O + n \times v}$$

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

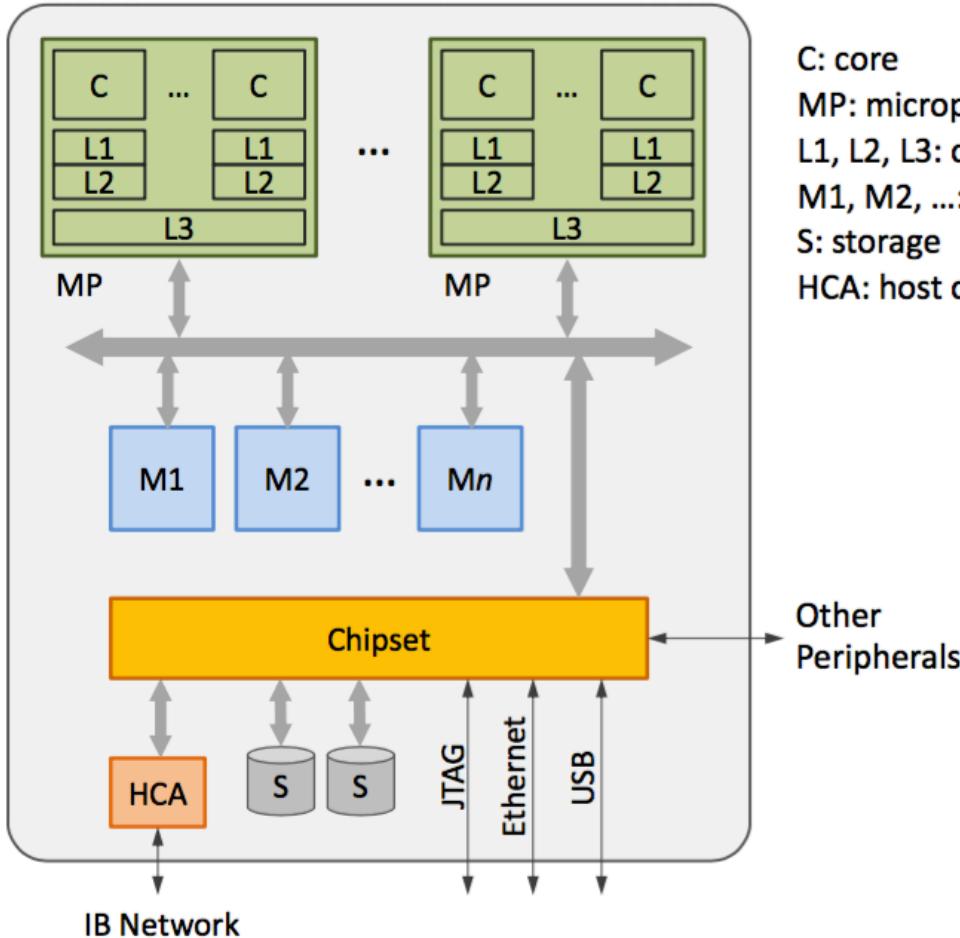


# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- **SMP System structure**
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# SMP Node Diagram



C: core

MP: microprocessor

L1, L2, L3: caches

M1, M2, ...: memory banks

S: storage

HCA: host channel adapter

Other  
Peripherals

# SMP System Examples

**Table 6.1 Some Examples of SMPs and Their Characteristics**

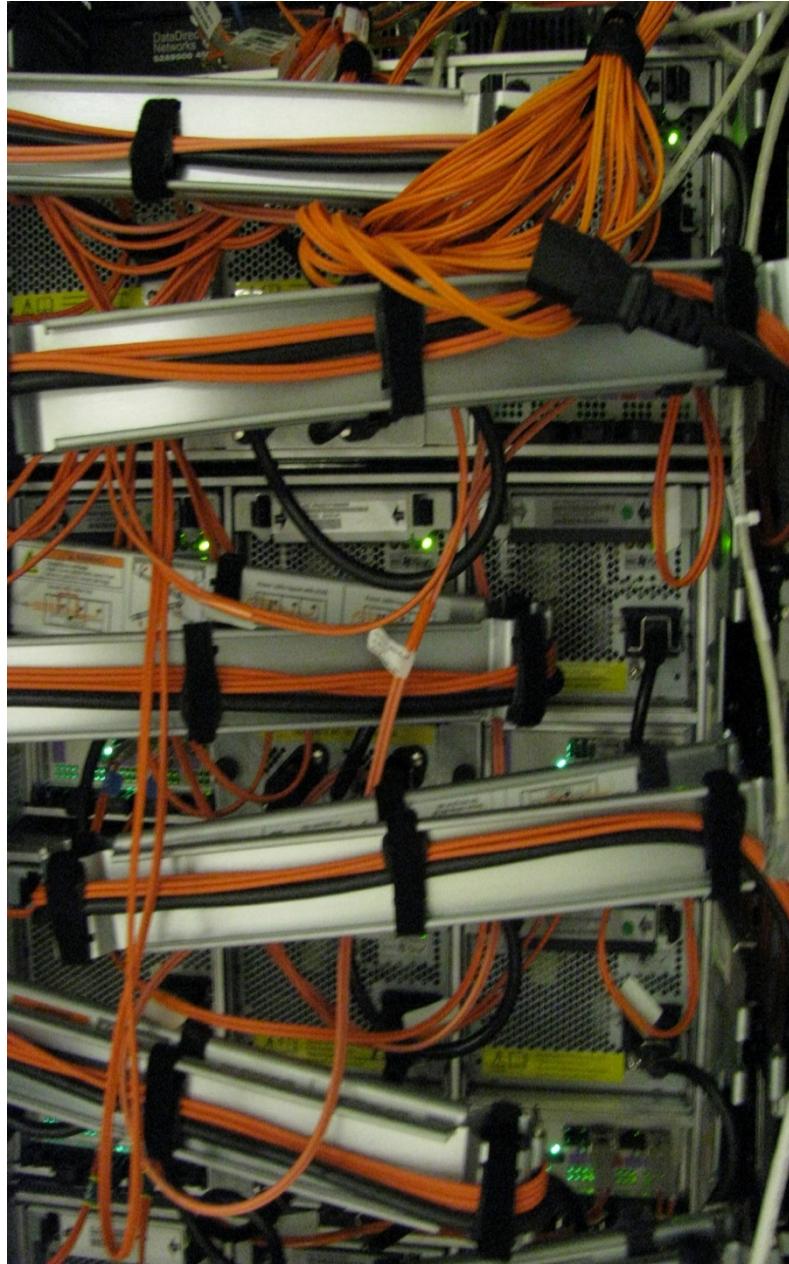
Vendor and Name	Processor	Number of Cores	Cores per Central Processing Unit	Memory Capacity	PCIe Slots	Storage Slots
IBM S822LC	IBM POWER8 2.92 GHz	20	10	256 GB	2 × 16-lane Gen.3 3 × 8-lane Gen.3	12 LFF
HPE rx2800 i6	Intel Itanium 9760 2.66 GHz	16	8	384 GB	3 × 16-lane Gen.2 2 × 8-lane Gen.2	8 SFF
Dell PowerEdge R930	Intel E7-8870v4 2.1 GHz	80	20	12 TB	10 Gen.3	24 SFF 8 NVMe
Oracle SPARC T7-4	SPARC M7 4.13 GHz	128	32	4 TB	8 × 16-lane Gen.3 8 × 8-lane Gen.3	8 SFF
HPE ProLiant DL385p Gen8	AMD Opteron 6373 2.3 GHz	32	16	384 GB	3 × 16-lane Gen.2 3 × 8-lane Gen.2	8 SFF

# Major Elements of an SMP Node

- Processor chip
- DRAM main memory cards
- Motherboard chip set
- On-board memory network
  - North bridge
- On-board I/O network
  - South bridge
- PCI industry standard interfaces
  - PCI, PCI-X, PCI-express
- System Area Network controllers
  - e.g. Ethernet, Myrinet, Infiniband, Quadrics, Federation Switch
- System Management network
  - Usually Ethernet
  - JTAG for low level maintenance
- Internal disk and disk controller
- Peripheral interfaces

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# Multicore Microprocessor Component Elements

- Multiple processor cores
  - One or more processors
- L1 caches
  - Instruction cache
  - Data cache
- L2 cache
  - Joint instruction/data cache
  - Dedicated to individual core processor
- L3 cache
  - Not all systems
  - Shared among multiple cores
  - Often off die but in same package
- Memory interface
  - Address translation and management (sometimes)
  - North bridge
- I/O interface
  - South bridge

# Comparison of Current Microprocessors

**Table 6.2 Characterization of Several SMP Processors**

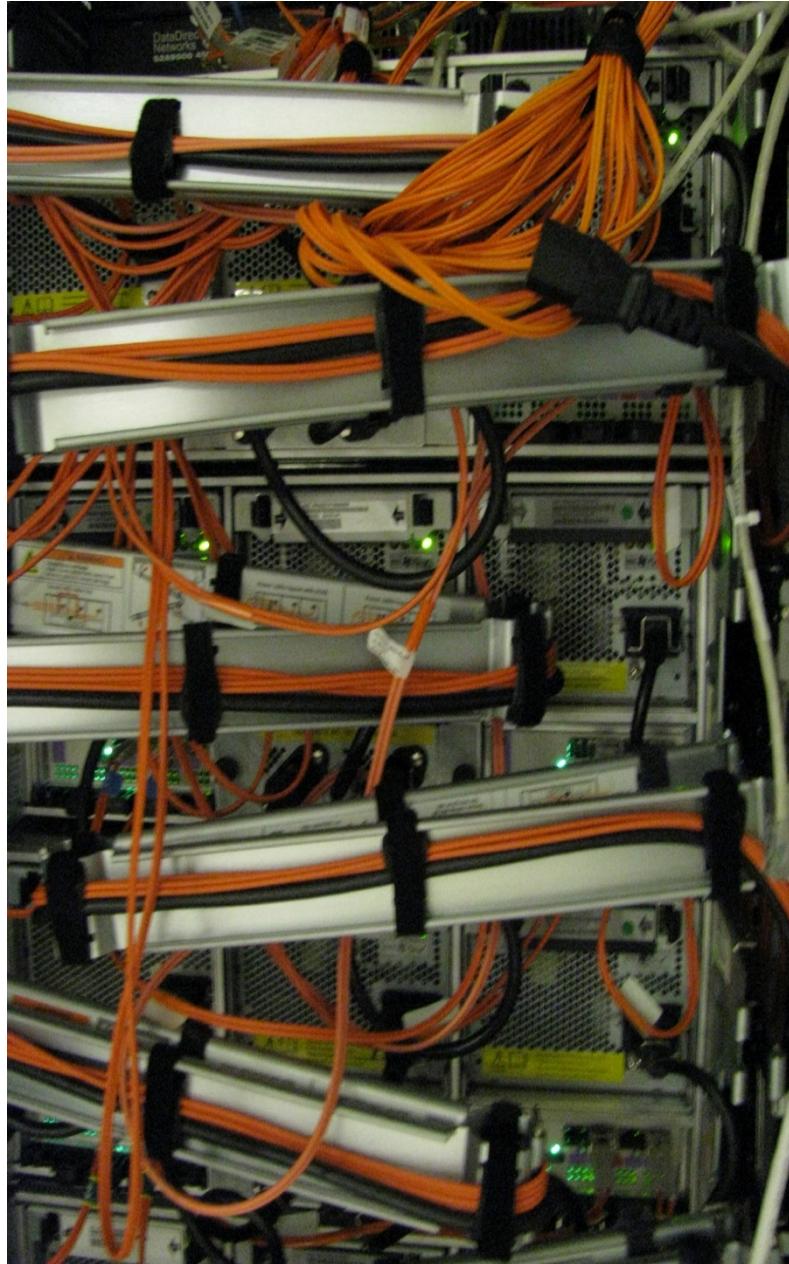
Processor	Clock Rate	Caches (per Core)	ILP (Each Core)	Cores Per Chip	Process and Die Size	Power (W)
AMD Opteron 6380	2.5 GHz (3.4 GHz turbo)	L1I: 32 KB L1D: 16 KB L2: 1 MB L3: 16 MB total	4 FPop/ cycle 4 intops/ cycle	16	32 nm, 316 mm <sup>2</sup>	115
IBM Power8	3.126 GHz (3.625 GHz turbo)	L1I: 64 KB L1D: 32 KB L2: 512 KB L3: 8 MB L4: 64 MB total	16 FPop/ cycle	12	22 nm, 650 mm <sup>2</sup>	190/247
Intel Xeon E7-8894V4	2.4 GHz (3.4 GHz turbo)	L1I: 32 KB L1D: 32 KB L2: 256 KB L3: 60 MB total	16 FPop/ cycle	24	14 nm	165

# Processor Core Micro Architecture

- Execution Pipeline
  - Stages of functionality to process issued instructions
  - Hazards are conflicts with continued execution
  - Forwarding supports closely associated operations exhibiting precedence constraints
- Out of Order Execution
  - Uses reservation stations
  - hides some core latencies and provide fine grain asynchronous operation supporting concurrency
- Branch Prediction
  - Permits computation to proceed at a conditional branch point prior to resolving predicate value
  - Overlaps follow-on computation with predicate resolution
  - Requires roll-back or equivalent to correct false guesses
  - Sometimes follows both paths, and several deep

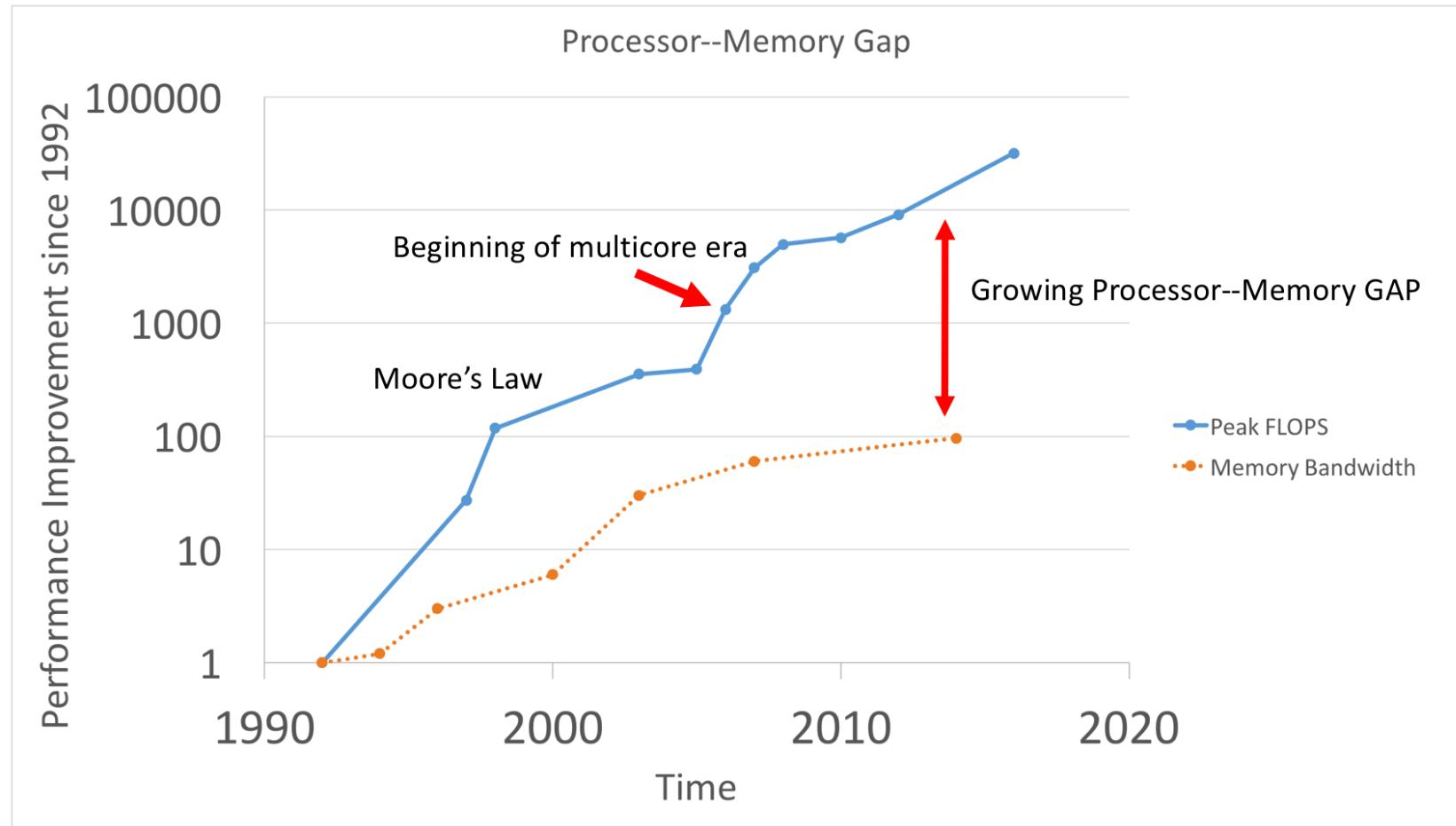
# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- **Memory System**
- Chip set
- South Bridge – I/O
- Performance Issues



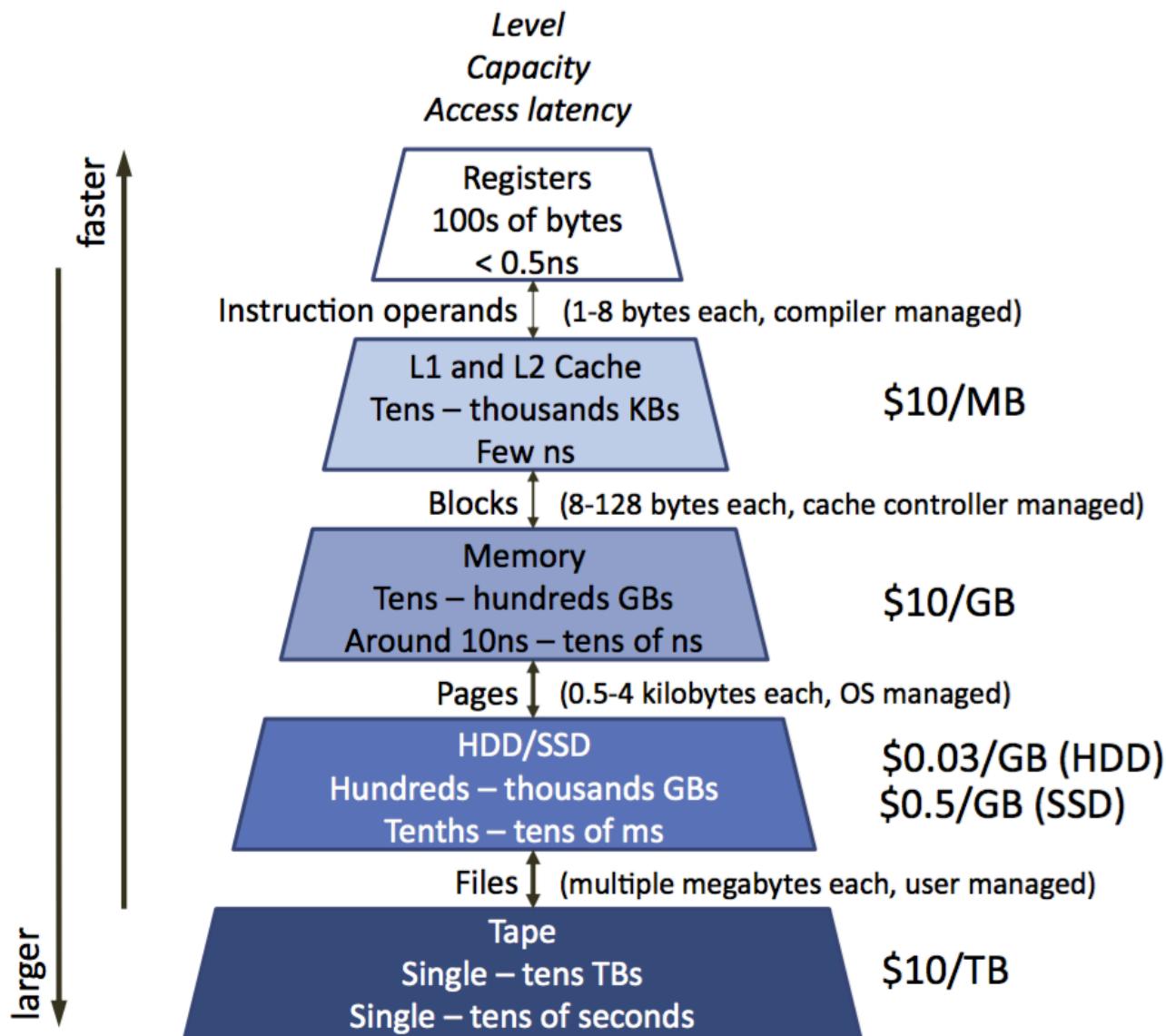
# Recap: Who Cares About the Memory Hierarchy?

## *Processor-DRAM Memory Gap (latency)*



# What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits spatial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers: a cache on variables
  - First-level cache: a cache on second-level cache
  - Second-level cache: a cache on memory
  - Memory: a cache on disk (virtual memory)
  - TLB :a cache on page table
  - Branch-prediction: a cache on prediction information

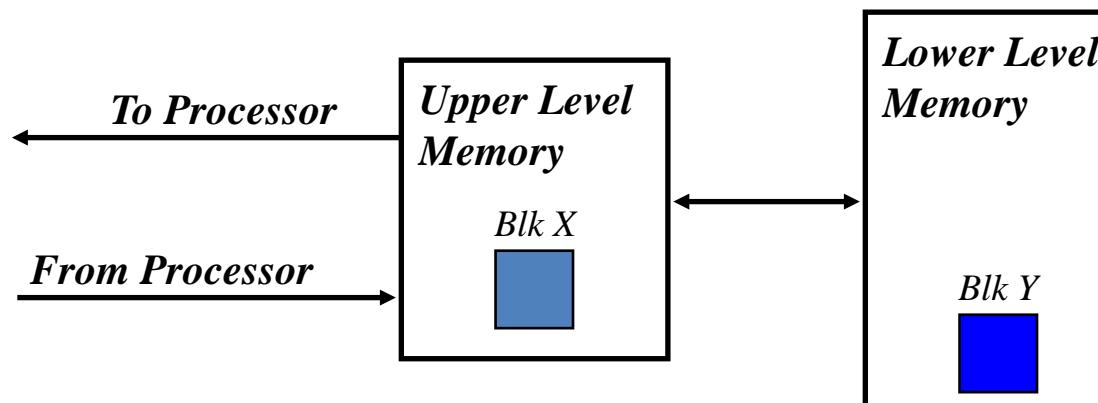


# Cache Measures

- **Hit rate:** fraction found in that level
  - So high that usually talk about **Miss rate**
- Average memory-access time
  - = Hit time + Miss rate x Miss penalty  
(ns or clocks)
- **Miss penalty:** time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    - =  $f(\text{latency to lower level})$
  - *transfer time*: time to transfer block
    - =  $f(BW \text{ between upper \& lower levels})$

# Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory accesses found in the upper level
  - **Hit Time**: Time to access the upper level which consists of  
RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** =  $1 - (\text{Hit Rate})$
  - **Miss Penalty**: Time to replace a block in the upper level +  
Time to deliver the block to the processor
- Hit Time  $\ll$  Miss Penalty (500 instructions on 21264!)



# Cache Performance

$$T = I_{count} \times CPI \times T_{cycle}$$

$$I_{count} = I_{ALU} + I_{MEM}$$

$$CPI = \left( \frac{I_{ALU}}{I_{count}} \right) \times CPI_{ALU} + \left( \frac{I_{MEM}}{I_{count}} \right) \times CPI_{MEM}$$

$T$  = total execution time

$T_{cycle}$  = time for a single processor cycle

$I_{count}$  = total number of instructions

$I_{ALU}$  = number of ALU instructions (e.g. register – register)

$I_{MEM}$  = number of memory access instructions ( e.g. load, store)

$CPI$  = average cycles per instructions

$CPI_{ALU}$  = average cycles per ALU instructions

$CPI_{MEM}$  = average cycles per memory instruction

$r_{miss}$  = cache miss rate

$r_{hit}$  = cache hit rate

$CPI_{MEM-MISS}$  = cycles per cache miss

$CPI_{MEM-HIT}$  =cycles per cache hit

$M_{ALU}$  = instruction mix for ALU instructions

$M_{MEM}$  = instruction mix for memory access instruction

# Cache Performance

*InstructionMix :*

$$M_{ALU} = \frac{I_{ALU}}{I_{count}}$$

$$M_{MEM} = \frac{I_{MEM}}{I_{count}}$$

$$M_{ALU} + M_{MEM} = 1$$

$$CPI = (M_{ALU} \times CPI_{ALU}) + (M_{MEM} \times CPI_{MEM})$$

$$T = I_{count} \times [(M_{ALU} \times CPI_{ALU}) + (M_{MEM} \times CPI_{MEM})] \times T_{cycle}$$

*T = total execution time*

*T<sub>cycle</sub> = time for a single processor cycle*

*I<sub>count</sub> = total number of instructions*

*I<sub>ALU</sub> = number of ALU instructions (e.g. register – register)*

*I<sub>MEM</sub> = number of memory access instructions ( e.g. load, store)*

*CPI = average cycles per instructions*

*CPI<sub>ALU</sub> = average cycles per ALU instructions*

*CPI<sub>MEM</sub> = average cycles per memory instruction*

*r<sub>miss</sub> = cache miss rate*

*r<sub>hit</sub> = cache hit rate*

*CPI<sub>MEM-MISS</sub> = cycles per cache miss*

*CPI<sub>MEM-HIT</sub> = cycles per cache hit*

*M<sub>ALU</sub> = instruction mix for ALU instructions*

*M<sub>MEM</sub> = instruction mix for memory access instruction*

# Cache Performance

$$CPI_{MEM} = CPI_{MEM-HIT} + r_{MISS} \times CPI_{MEM-MISS}$$

$$T = I_{count} \times [(M_{ALU} \times CPI_{ALU}) + (M_{MEM} \times (CPI_{MEM-HIT} + r_{MISS} \times CPI_{MEM-MISS}))) \times T_{cycle}]$$

$T$  = total execution time

$T_{cycle}$  = time for a single processor cycle

$I_{count}$  = total number of instructions

$I_{ALU}$  = number of ALU instructions (e.g. register – register)

$I_{MEM}$  = number of memory access instructions ( e.g. load, store)

$CPI$  = average cycles per instructions

$CPI_{ALU}$  = average cycles per ALU instructions

$CPI_{MEM}$  = average cycles per memory instruction

$r_{miss}$  = cache miss rate

$r_{hit}$  = cache hit rate

$CPI_{MEM-MISS}$  = cycles per cache miss

$CPI_{MEM-HIT}$  =cycles per cache hit

$M_{ALU}$  = instruction mix for ALU instructions

$M_{MEM}$  = instruction mix for memory access instruction

# Cache Performance: Example

$$I_{count} = 10^{11}$$

$$I_{MEM} = 2 \times 10^{10}$$

$$CPI_{ALU} = 1$$

$$T_{cycle} = 0.5\text{ns}$$

$$CPI_{MEM-MISS} = 100$$

$$CPI_{MEM-HIT} = 1$$

$$I_{ALU} = I_{count} - I_{MEM} = 8 \times 10^{10}$$

$$M_{ALU} = \frac{I_{ALU}}{I_{count}} = \frac{8 \times 10^{10}}{10^{11}} = \frac{8}{10} = 0.8$$

$$M_{MEM} = \frac{I_{MEM}}{I_{count}} = \frac{2 \times 10^{10}}{10^{11}} = 0.2$$

$$r_{hitA} = 0.9$$

$$\begin{aligned} CPI_{MEM-A} &= CPI_{MEM-HIT} + r_{MISS-A} \times CPI_{MEM-MISS} \\ &= 1 + (1 - 0.9) \times 100 = 11 \end{aligned}$$

$$\begin{aligned} T_A &= 10^{11} \times ((0.8 \times 1) + (0.2 \times 11)) \times 5 \times 10^{-10} \\ &= 150 \text{ sec} \end{aligned}$$

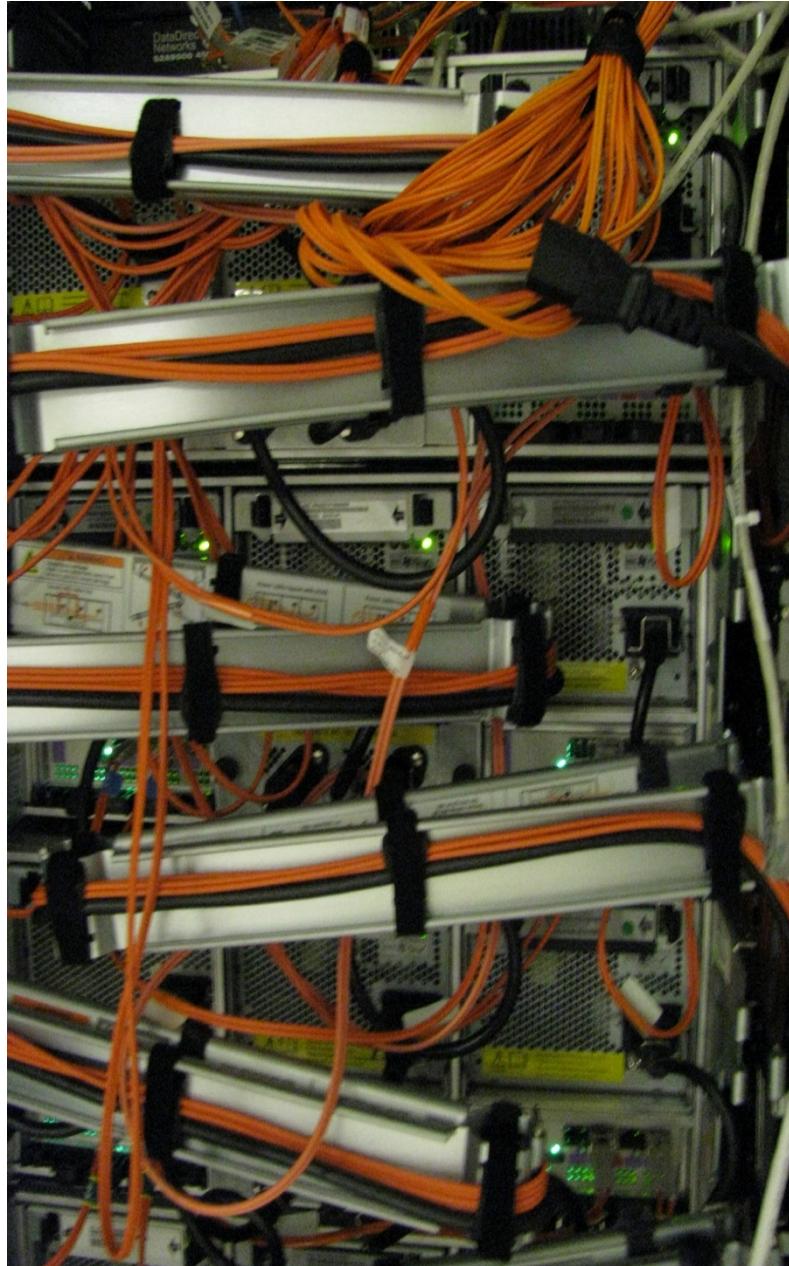
$$r_{hitB} = 0.5$$

$$\begin{aligned} CPI_{MEM-B} &= CPI_{MEM-HIT} + r_{MISS-B} \times CPI_{MEM-MISS} \\ &= 1 + (1 - 0.5) \times 100 = 51 \end{aligned}$$

$$\begin{aligned} T_B &= 10^{11} \times ((0.8 \times 1) + (0.2 \times 51)) \times 5 \times 10^{-10} \\ &= 550 \text{ sec} \end{aligned}$$

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues



# Motherboard Chipset

- Provides core functionality of motherboard
- Embeds low-level protocols to facilitate efficient communication between local components of computer system
- Controls the flow of data between the CPU, system memory, on-board peripheral devices, expansion interfaces and I/O subsystem
- Also responsible for power management features, retention of non-volatile configuration data and real-time measurement
- Typically consists of:
  - Northbridge (Memory Controller Hub, MCH), managing traffic between the processor, RAM, GPU, southbridge and optionally PCI Express slots
  - Southbridge (I/O Controller Hub, ICH), coordinating slower set of devices, including traditional PCI bus, ISA bus, SMBus, IDE (ATA), DMA and interrupt controllers, real-time clock, BIOS memory, ACPI power management, LPC bridge (providing fan control, floppy disk, keyboard, mouse, MIDI interfaces, etc.), and optionally Ethernet, USB, IEEE1394, audio codecs and RAID interface

# Major Chipset Vendors

- Intel
  - <http://developer.intel.com/products/chipsets/index.htm>
- Via
  - <http://www.via.com.tw/en/products/chipsets>
- SiS
  - [http://www.sis.com/products/product\\_000001.htm](http://www.sis.com/products/product_000001.htm)
- AMD/ATI
  - <http://ati.amd.com/products/integrated.html>
- Nvidia
  - <http://www.nvidia.com/page/mobo.html>

# Motherboard

- Also referred to as main board, system board, backplane
- Provides mechanical and electrical support for pluggable components of a computer system
- Constitutes the central circuitry of a computer, distributing power and clock signals to target devices, and implementing communication backplane for data exchanges between them
- Defines expansion possibilities of a computer system through slots accommodating special purpose cards, memory modules, processor(s) and I/O ports
- Available in many form factors and with various capabilities to match particular system needs, housing capacity and cost

# Motherboard Form Factors

- Refer to standardized motherboard sizes
- Most popular form factor used today is ATX, evolved from now obsolete AT (Advanced Technology) format
- Examples of other common form factors:
  - MicroATX, miniaturized version of ATX
  - WTX, large form factor designated for use in high power workstations/servers featuring multiple processors
  - Mini-ITX, designed for use in thin clients
  - PC/104 and ETX, used in embedded systems and single board computers
  - BTX (Balanced Technology Extended), introduced by Intel as a possible successor to ATX

# Motherboard Manufacturers

- Abit
- Albatron
- Aopen
- ASUS
- Biostar
- DFI
- ECS
- Epox
- FIC
- Foxconn
- Gigabyte
- IBM
- Intel
- Jetway
- MSI
- Shuttle
- Soyo
- SuperMicro
- Tyan
- VIA

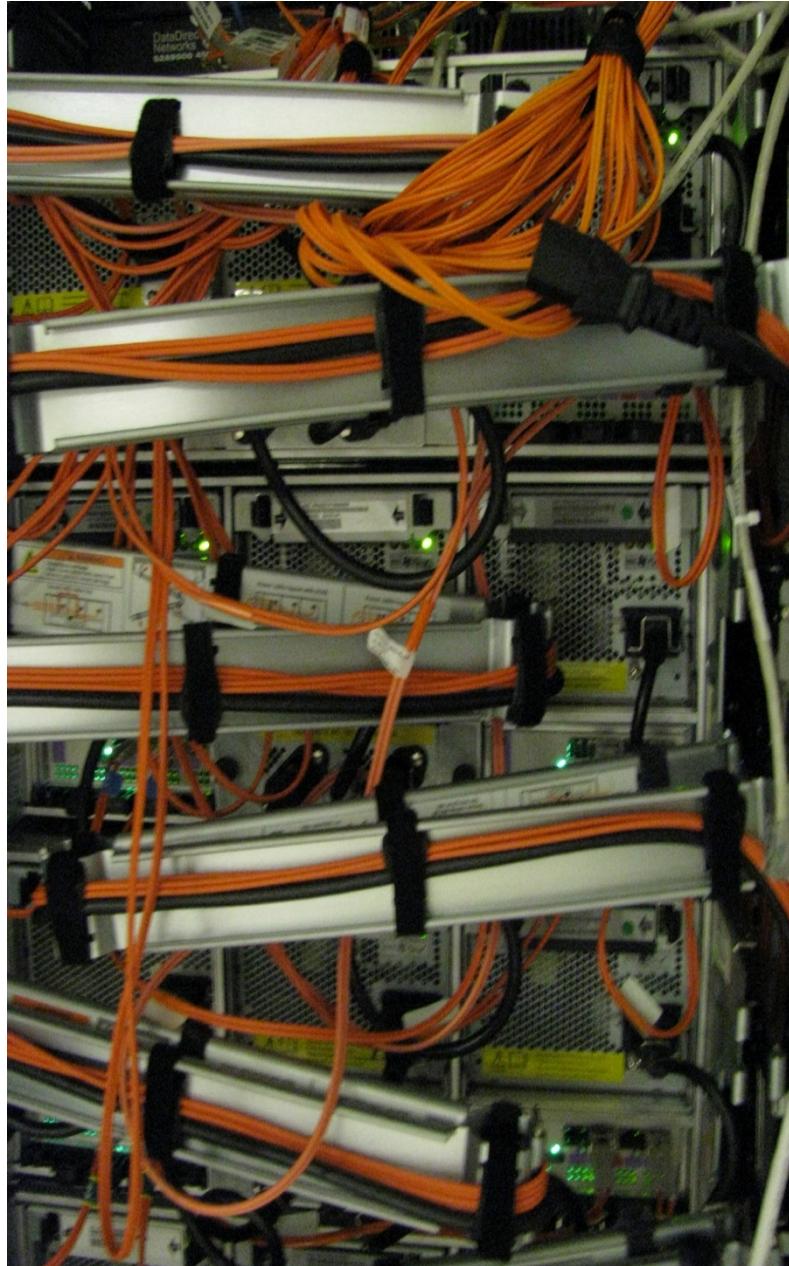
# Populated CPU Socket



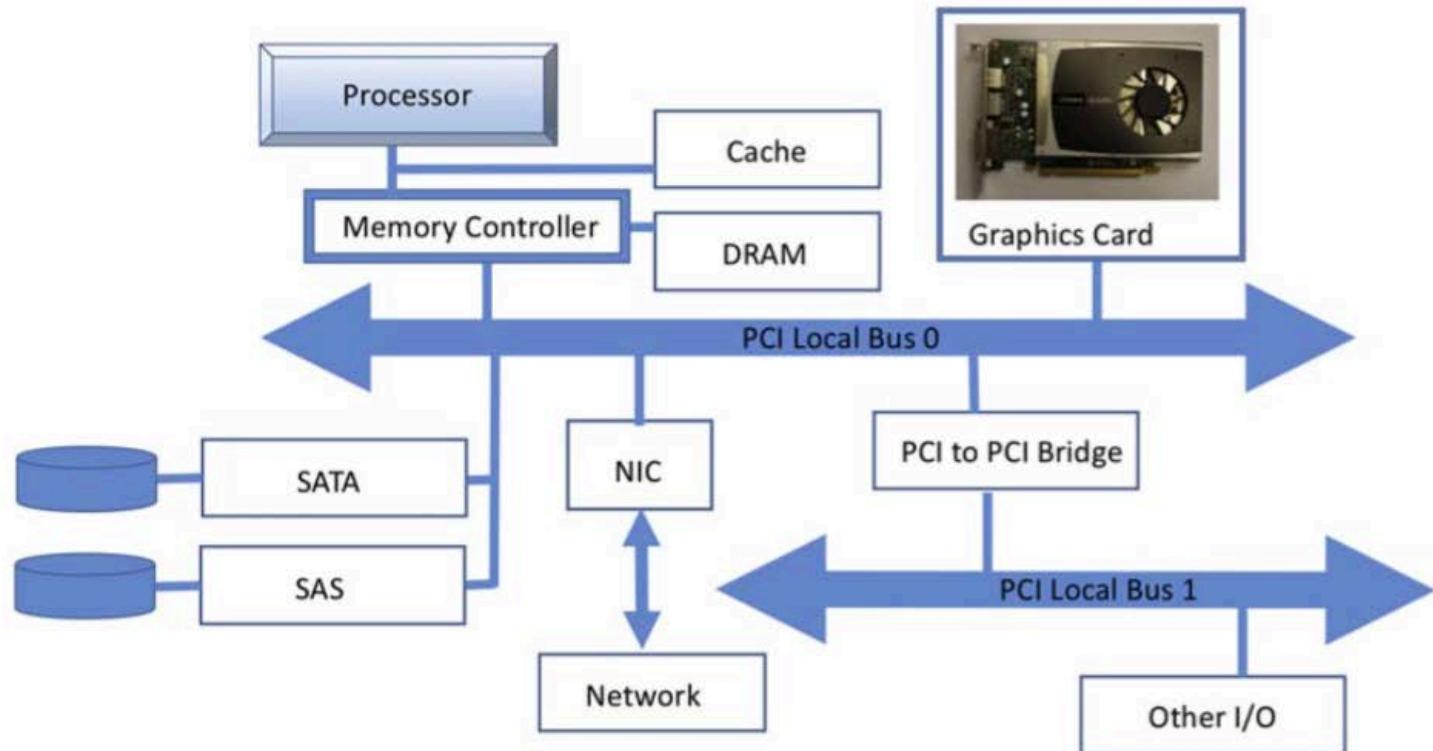
Source: <http://www.motherboards.org>

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- Performance Issues

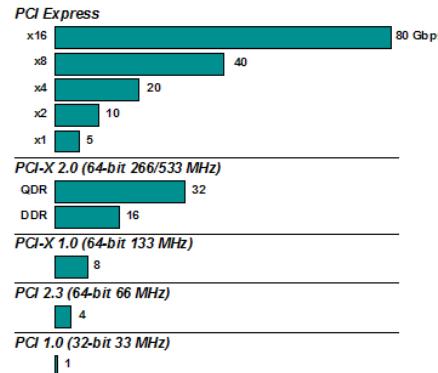


## Layout of system equipped with multiple PCI buses



# PCI-express

Lane width	Clock speed	Throughput (duplex, bits)	Throughput (duplex, bytes)	Initial expected uses
x1	2.5 GHz	5 Gbps	400 MBps	Slots, Gigabit Ethernet
x2	2.5 GHz	10 Gbps	800 MBps	
x4	2.5 GHz	20 Gbps	1.6 GBps	Slots, 10 Gigabit Ethernet, SCSI, SAS
x8	2.5 GHz	40 Gbps	3.2 GBps	
x16	2.5 GHz	80 Gbps	6.4 GBps	Graphics adapters

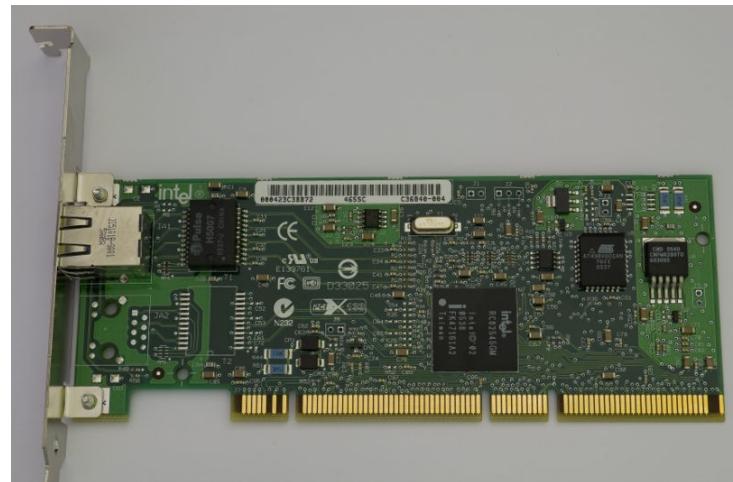


# PCI-X

	<b>Bus Width</b>	<b>Clock Speed</b>	<b>Features</b>	<b>Bandwidth</b>
PCI-X 66	64 Bits	66 MHz	Hot Plugging, 3.3 V	533 MB/s
PCI-X 133	64 Bits	133 MHz	Hot Plugging, 3.3 V	1.06 GB/s
PCI-X 266	64 Bits, optional 16 Bits only	133 MHz Double Data Rate	Hot Plugging, 3.3 & 1.5 V, ECC supported	2.13 GB/s
PCI-X 533	64 Bits, optional 16 Bits only	133 MHz Quad Data Rate	Hot Plugging, 3.3 & 1.5 V, ECC supported	4.26 GB/s



Mode	V/I/O	64-Bit		32-Bit		16-Bit
		Slots	MB	Slots	MB	
PCI 33	5V/3.3V	266		133		N/A
PCI 66*	3.3V	533		266		N/A
PCI-X 66	3.3V	533		266		N/A
PCI-X 133 (operating at 100 MHz)	3.3V	800		400		N/A
PCI-X 133	3.3V	1066		533		N/A
PCI-X 266	1.5V	2133		1066	533	
PCI-X 533	1.5V	4266		2133	1066	

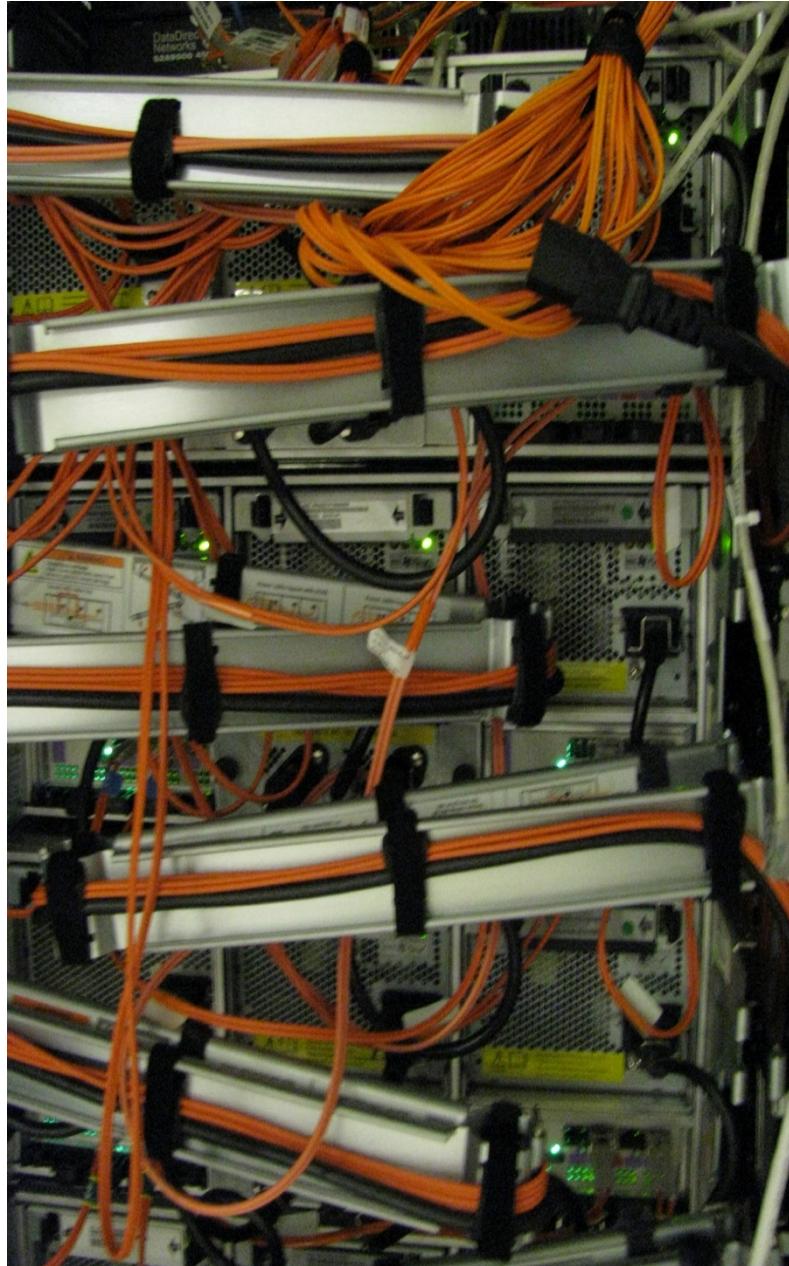


# Bandwidth Comparisons

CONNECTION	BITS	BYTES
<a href="#">PCI</a> 32-bit/33 MHz	1.06666 Gbit/s	<b>133.33 MB/s</b>
<a href="#">PCI</a> 64-bit/33 MHz	2.13333 Gbit/s	<b>266.66 MB/s</b>
<a href="#">PCI</a> 32-bit/66 MHz	2.13333 Gbit/s	<b>266.66 MB/s</b>
<a href="#">PCI</a> 64-bit/66 MHz	4.26666 Gbit/s	<b>533.33 MB/s</b>
<a href="#">PCI</a> 64-bit/100 MHz	6.39999 Gbit/s	<b>799.99 MB/s</b>
<a href="#">PCI Express</a> (x1 link) <sup>[6]</sup>	2.5 Gbit/s	<b>250 MB/s</b>
<a href="#">PCI Express</a> (x4 link) <sup>[6]</sup>	10 Gbit/s	<b>1 GB/s</b>
<a href="#">PCI Express</a> (x8 link) <sup>[6]</sup>	20 Gbit/s	<b>2 GB/s</b>
<a href="#">PCI Express</a> (x16 link) <sup>[6]</sup>	40 Gbit/s	<b>4 GB/s</b>
<a href="#">PCI Express</a> 2.0 (x32 link) <sup>[6]</sup>	80 Gbit/s	<b>8 GB/s</b>
<a href="#">PCI-X</a> DDR 16-bit	4.26666 Gbit/s	<b>533.33 MB/s</b>
<a href="#">PCI-X</a> 133	8.53333 Gbit/s	<b>1.06666 GB/s</b>
<a href="#">PCI-X</a> QDR 16-bit	8.53333 Gbit/s	<b>1.06666 GB/s</b>
<a href="#">PCI-X</a> DDR	17.066 Gbit/s	<b>2.133 GB/s</b>
<a href="#">PCI-X</a> QDR	34.133 Gbit/s	<b>4.266 GB/s</b>
<a href="#">AGP</a> 8x	17.066 Gbit/s	<b>2.133 GB/s</b>

# Topics

- Introduction
- SMP Context
- Performance: Amdahl's Law
- SMP System structure
- Processor core
- Memory System
- Chip set
- South Bridge – I/O
- **Performance Issues**



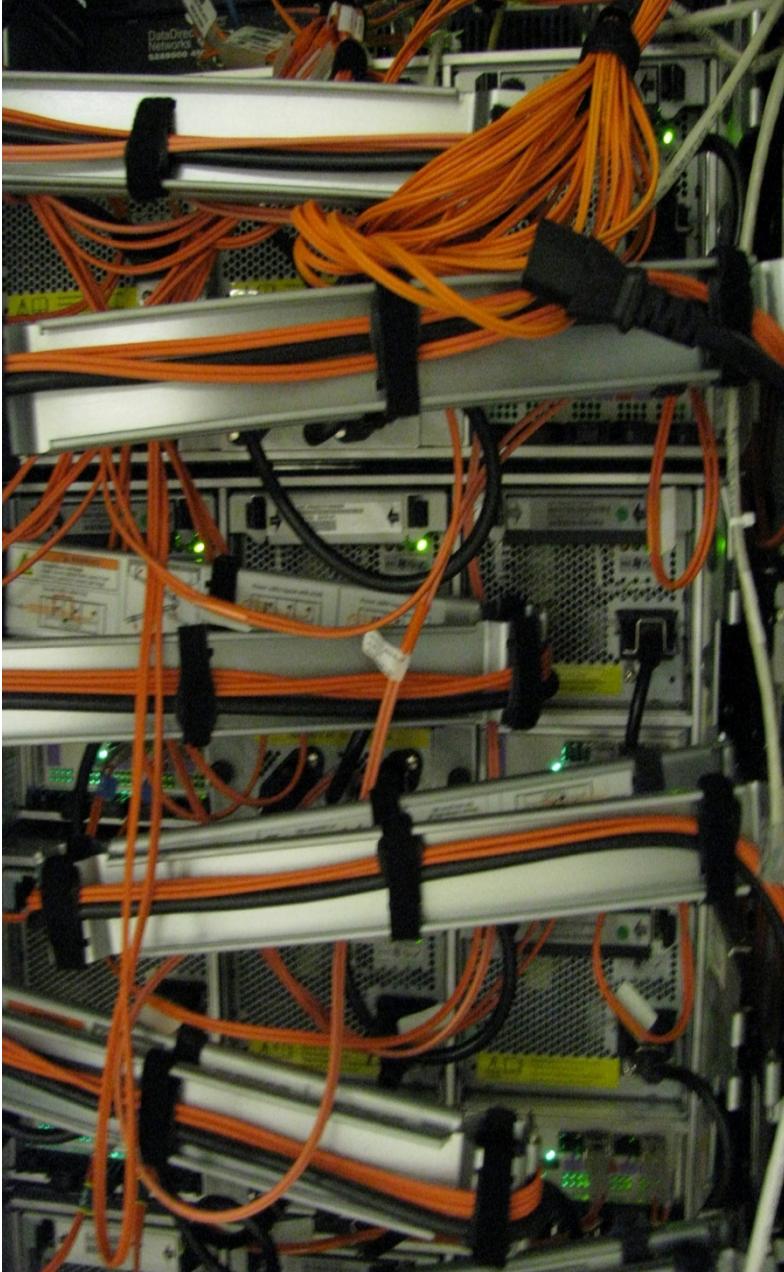
# Performance Issues

- Cache behavior
  - Hit/miss rate
  - Replacement strategies
- Prefetching
- Clock rate
- ILP
- Branch prediction
- Memory
  - Access time
  - Bandwidth

# The Essential OpenMP

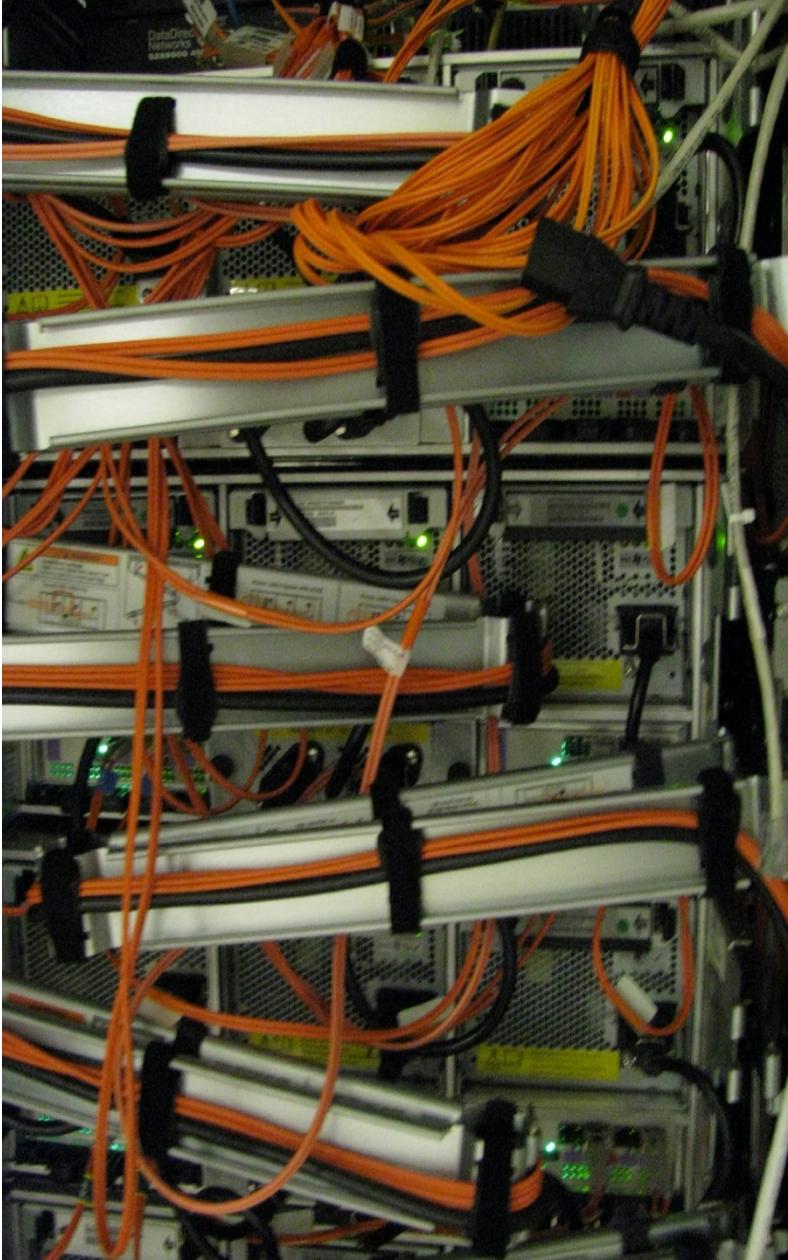
# Lecture 6: OpenMP

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# Where are we? (Take a deep breath...)

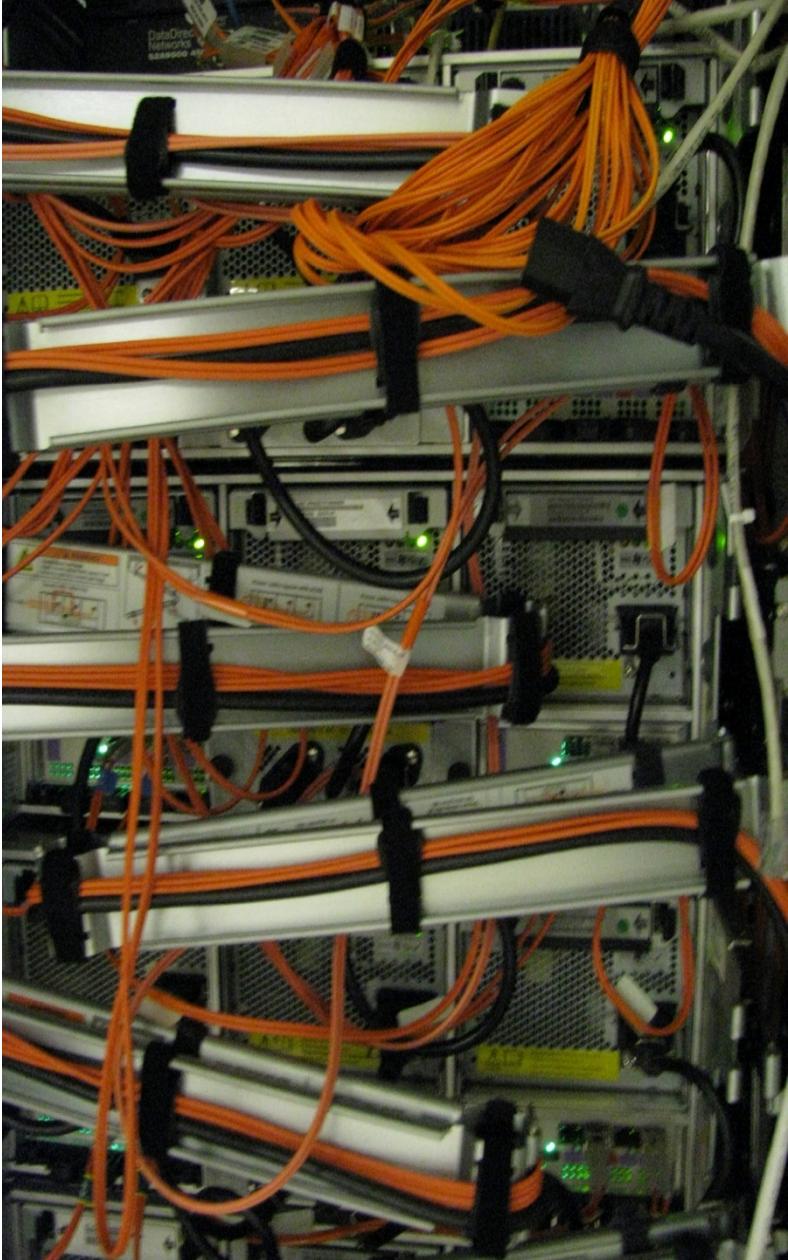
- 3 classes of parallel/distributed computing
  - Capacity
  - Capability
  - Cooperative
- 3 classes of parallel architectures (respectively)
  - Loosely coupled clusters and workstation farms
  - Tightly coupled vector, SIMD, SMP
  - Distributed memory MPPs (and some clusters)
- 3 classes of parallel execution models (respectively)
  - Workflow, throughput, SPMD (ssh)
  - Multithreaded with shared memory semantics (Pthreads)
  - Communicating Sequential Processes (sockets)
- 3 classes of programming models
  - Condor (Segment 1)
  - OpenMP (Segment 3)
  - MPI (Segment 2)

# HPC Modalities

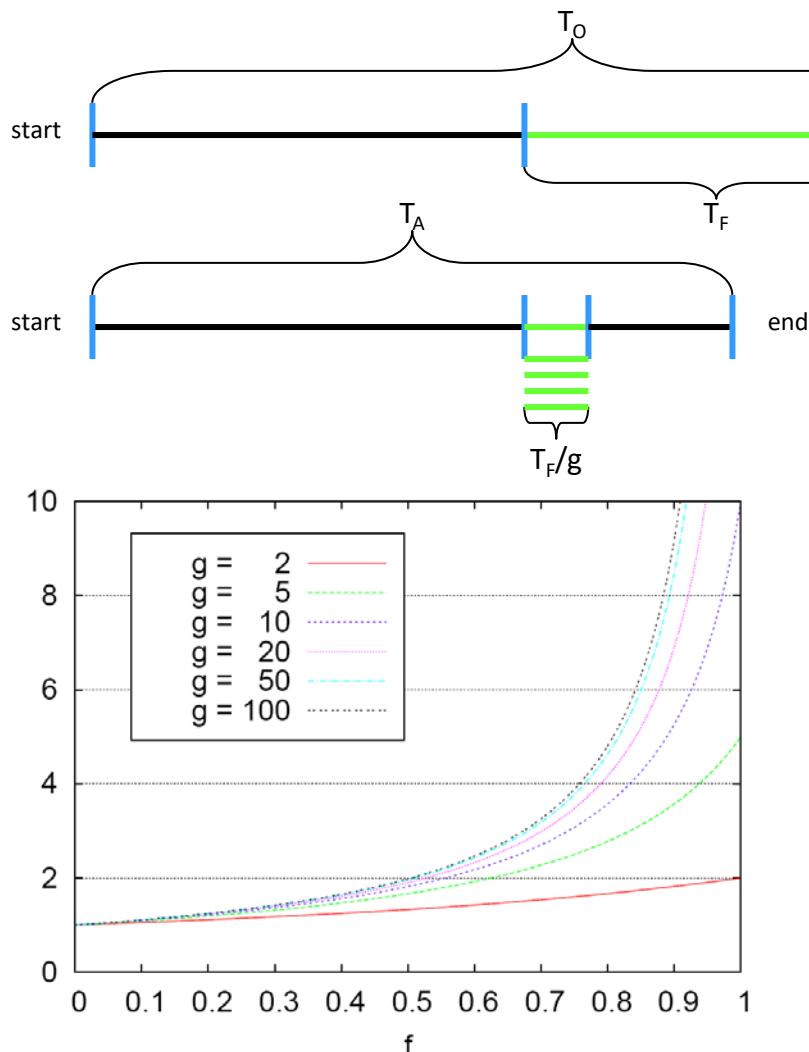
Modalities	Degree of Integration	Architectures	Execution Models	Programming Models
Capacity	Loosely Coupled	Clusters & Workstation farms	Workflow Throughput	Condor
Capability	Tightly Coupled	Vectors, SMP, SIMD	Shared Memory Multithreading	OpenMP
Cooperative	Medium	DM MPPs & Clusters	CSP	MPI

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# 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

$S \equiv$  speedup of computation with acceleration applied

$$S = T_o / T_A$$

$$f = T_F / T_o$$

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

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

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

# Performance: Caches & Locality

- **Temporal Locality** is a property that if a program accesses a memory location, there is a much higher than random probability that the same location would be accessed again.
- **Spatial Locality** is a property that if a program accesses a memory location, there is a much higher than random probability that the nearby locations would be accessed soon.
- Spatial locality is usually easier to achieve than temporal locality
- A couple of key factors affect the relationship between locality and scheduling :
  - Size of dataset being processed by each processor
  - How much reuse is present in the code processing a chunk of iterations.

# Performance Shared Memory (OpenMP): Key Factors

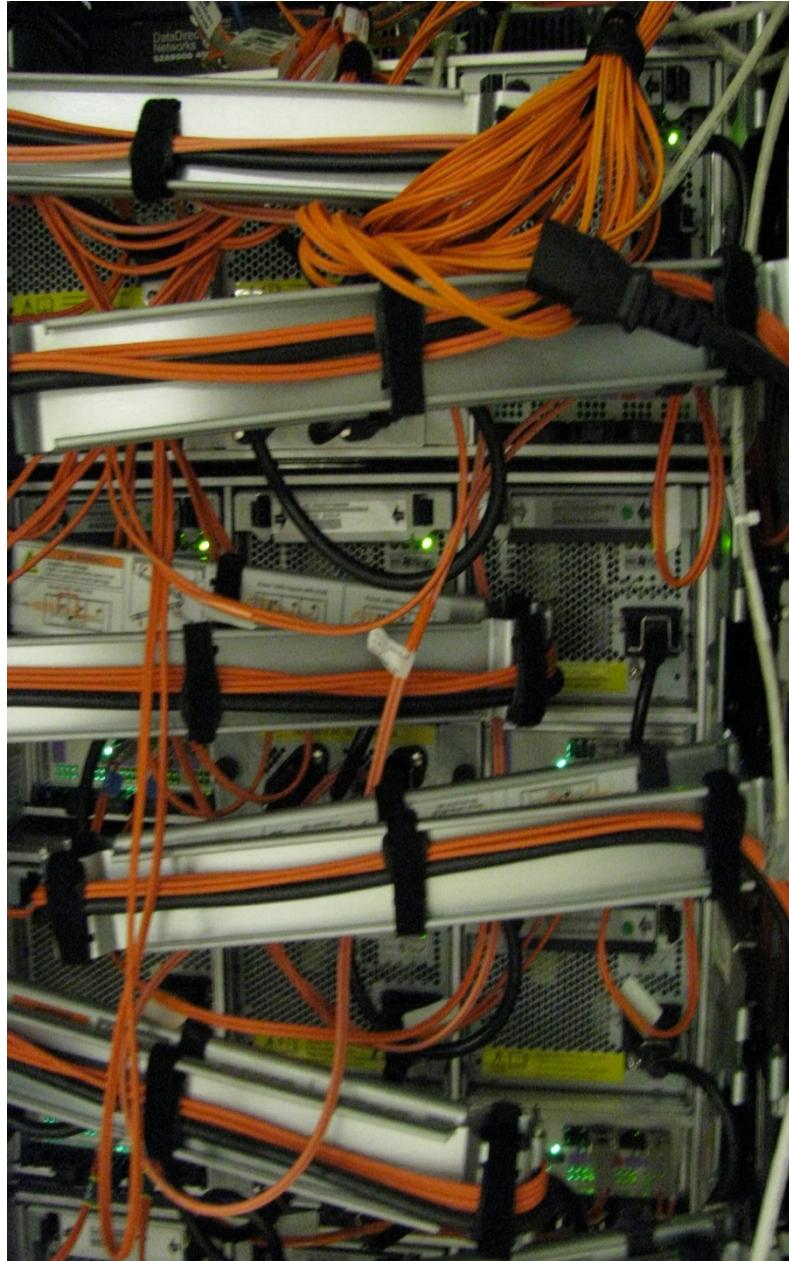
- Load Balancing :
  - mapping workloads with thread scheduling
- Caches :
  - Write-through
  - Write-back
- Locality :
  - Temporal Locality
  - Spatial Locality
- How Locality affects scheduling algorithm selection
- Synchronization :
  - Effect of critical sections on performance

# Performance: Caches & Locality

- Caches (Review) :
  - for a C statement :
    - $a[i] = b[i]+c[i]$
  - the system accesses the memory locations referenced by  $b[i]$  and  $c[i]$  to the processor, the result of the computation is subsequently stored in the memory location referenced by  $a[i]$
- **Write-through caches**: When a user writes some data, the data is immediately written back to the memory, thus maintaining the cache-memory consistency. In write through caches data in caches always reflect the data in the memory. One of the main issues in write through caches is the increase in system overhead required due to moving of large data between cache and memory.
- **Write-back caches** : When a user writes some data, the data is stored in the cache and is not synchronized with the memory. Instead when the cache content is different than the memory content, a bit entry is made in the cache. While cleaning up caches the system checks for the entry in cache and if the bit is set the system writes the changes to the memory.

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- **Introduction to OpenMP**
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# Introduction

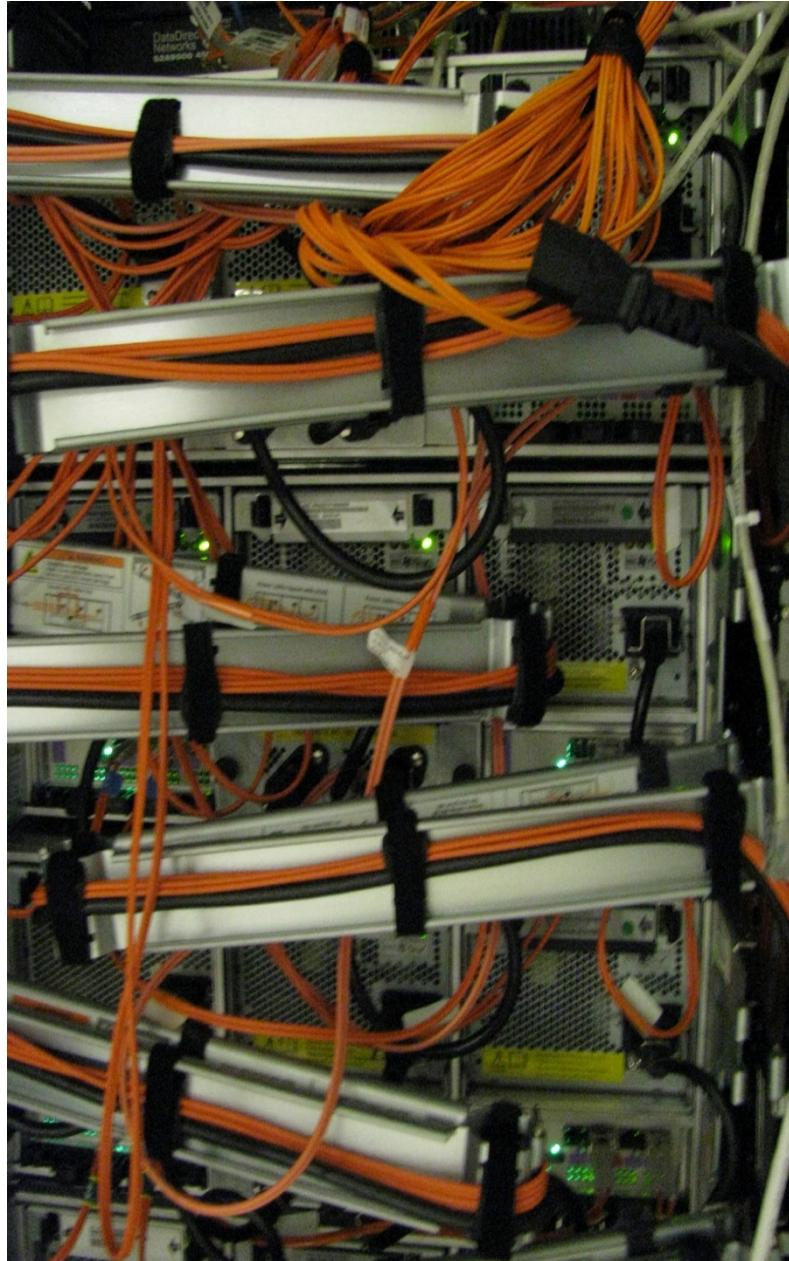
- OpenMP is :
  - an API (Application Programming Interface)
  - NOT a programming language
  - A set of compiler directives that help the application developer to parallelize their workload.
  - A collection of the directives, environment variables and the library routines
- OpenMP is composed of the following main components :
  - Directives
  - Runtime library routines
  - Environment variables

# Components of OpenMP

Directives	Runtime library routines	Environment variables
Parallel regions	Number of threads	Number of threads
Work sharing	Thread ID	Scheduling type
Synchronization	Dynamic thread adjustment	Dynamic thread adjustment
Data scope attributes : <ul style="list-style-type: none"><li>• private</li><li>• firstprivate</li><li>• last private</li><li>• shared</li><li>• reduction</li></ul>	Nested Parallelism	Nested Parallelism
Orphaning	Timers	
	API for locking	

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- **OpenMP: Runtime Library & Environment Variables**
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# Runtime Library Routines

- Runtime library routines help manage parallel programs
- Many runtime library routines have corresponding environment variables that can be controlled by the users
- Runtime libraries can be accessed by including **omp.h** in applications that use OpenMP : #include <omp.h>
- For example for calls like :
  - **omp\_get\_num\_threads()**, (by which an openMP program determines the number of threads available for execution) can be controlled using an environment variable set at the command-line of a shell  
**(\$OMP\_NUM\_THREADS)**
- Some of the activities that the OpenMP libraries help manage are :
  - Determining the number of threads/processors
  - Scheduling policies to be used
  - General purpose locking and portable wall clock timing routines

# OpenMP: Runtime Library

---

Function: `omp_get_num_threads()`

C/ C++ `int omp_get_num_threads(void);`

Fortran `integer function omp_get_num_threads()`

---

Description:

Returns the total number of threads currently in the group executing the parallel block from where it is called.

---

---

Function: `omp_get_thread_num()`

C/ C++ `int omp_get_thread_num(void);`

Fortran `integer function omp_get_thread_num()`

---

Description:

For the master thread, this function returns zero. For the child nodes the call returns an integer between `1` and `omp_get_num_threads()-1` inclusive.

---

# OpenMP Environment Variables

- OpenMP provides 4 main environment variables for controlling execution of parallel codes:
  - **OMP\_NUM\_THREADS** – controls the parallelism of the OpenMP application
  - **OMP\_DYNAMIC** – enables dynamic adjustment of number of threads for execution of parallel regions
  - **OMP\_SCHEDULE** – controls the load distribution in loops such as **do**, **for**
  - **OMP\_NESTED** – Enables nested parallelism in OpenMP applications

# OpenMP Environment Variables

---

Environment Variable: **OMP\_NUM\_THREADS**

Usage : **OMP\_NUM\_THREADS *n***  
bash/sh/ksh: *export OMP\_NUM\_THREADS=8*  
csh/tcsh *setenv OMP\_NUM\_THREADS 8*

Description:

Sets the number of threads to be used by the OpenMP program during execution.

---

Environment Variable: **OMP\_DYNAMIC**

Usage : **OMP\_DYNAMIC {TRUE|FALSE}**  
bash/sh/ksh: *export OMP\_DYNAMIC=TRUE*  
csh/tcsh *setenv OMP\_DYNAMIC “TRUE”*

Description:

When this environment variable is set to TRUE the maximum number of threads available for use by the OpenMP program is ***n*** (\$OMP\_NUM\_THREADS).

---

# OpenMP Environment Variables

---

Environment Variable: **OMP\_SCHEDULE**

Usage :	OMP_SCHEDULE “ <i>schedule,[chunk]</i> ”
bash/sh/ksh:	export OMP_SCHEDULE static,N/P
csh/tcsh	setenv OMP_SCHEDULE=“GUIDED,4”

Description:

Only applies to **for** and **parallel for** directives. This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number, the default being 1.

---

Environment Variable: **OMP\_NESTED**

Usage :	OMP_NESTED {TRUE FALSE}
bash/sh/ksh:	export OMP_NESTED FALSE
csh/tcsh	setenv OMP_NESTED=“FALSE”

Description:

Setting this environment variable to **TRUE** enables multi-threaded execution of inner parallel regions in nested parallel regions.

---

# OpenMP: Basic Constructs

## OpenMP Execution Model (FORK/JOIN):

Sequential Part (master thread)

Parallel Region (**FORK** : group of threads)

Sequential Part (**JOIN**: master thread)

Parallel Region (**FORK**: group of threads)

Sequential Part (**JOIN** : master thread)

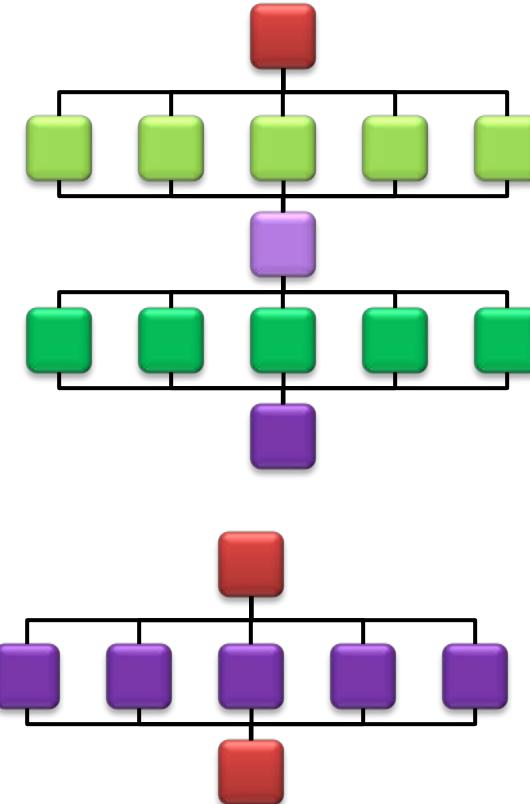
## C / C++ :

```
#pragma omp parallel {  
    parallel block  
} /* omp end parallel */
```

To invoke library routines in C/C++ add

```
#include <omp.h>
```

near the top of your code



# HelloWorld in OpenMP

```
#include <omp.h>

main ()
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

Non shared copies of  
data for each thread

OpenMP directive to  
indicate START  
segment to be  
parallelized

Code segment that  
will be executed in  
parallel

OpenMP directive to  
indicate END  
segment to be  
parallelized

# OpenMP Execution

- On encountering the C construct `#pragma omp parallel{`, **n-1** extra threads are created
- `omp_get_thread_num()` returns a unique identifier for each thread that can be utilized. The value returned by this call is between **0** and **(OMP\_NUM\_THREADS – 1)**
- `omp_get_num_threads()` returns the total number of threads involved in the parallel section of the program
- Code after the parallel directive is executed independently on each of the **n** threads.
- On encountering the C construct `}` (corresponding to `#pragma omp parallel{` ), indicates the end of parallel execution of the code segment, the **n-1** extra threads are deactivated and normal sequential execution begins.

# Compiling OpenMP Programs

## C :

- Case sensitive directives
- Syntax :
  - `#pragma omp directive [clause [clause]..]`
- Compiling OpenMP source code :
  - (**GNU C compiler**) : `gcc -fopenmp -o exec_name file_name.c`
  - (**Intel C compiler**) : `icc -o exe_file_name -openmp file_name.c`

## Fortran :

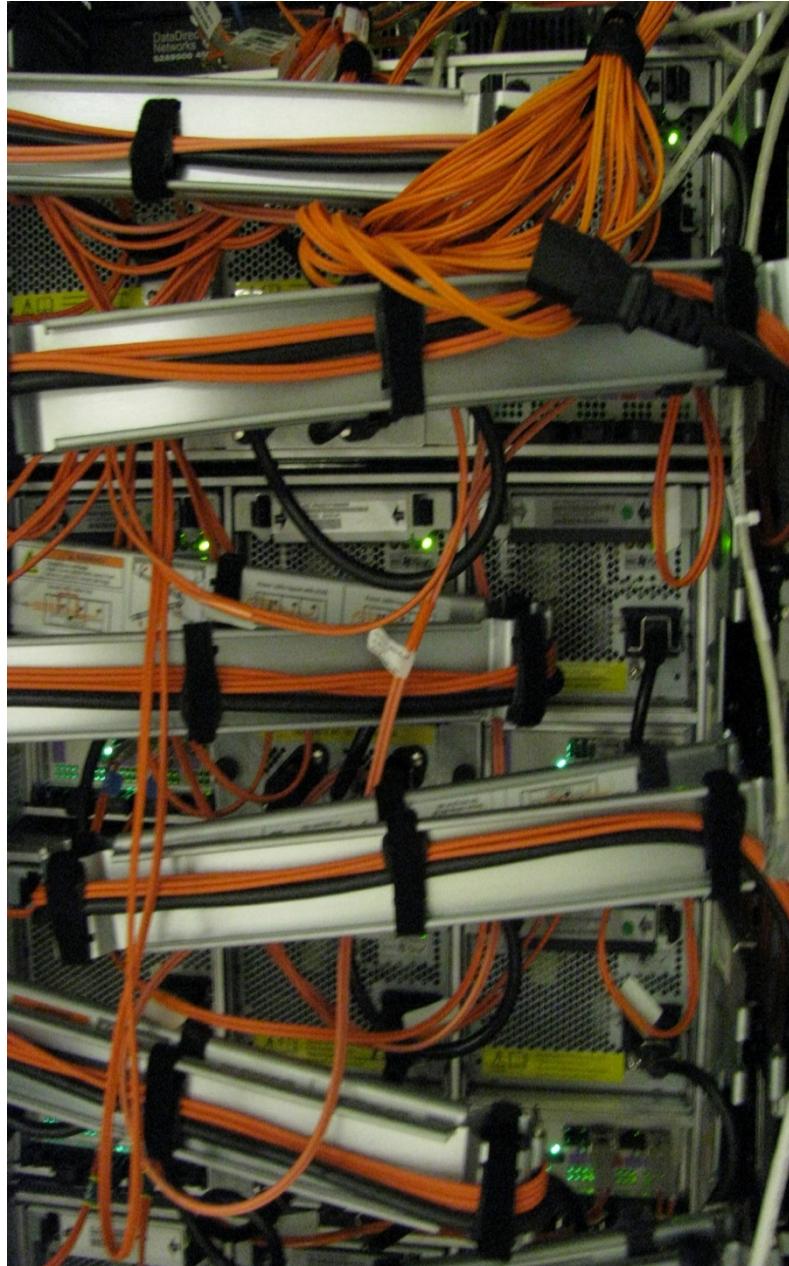
- Case insensitive directives
- Syntax :
  - !\$OMP directive [clause[,] clause]... (free format)
  - !\$OMP / C\$OMP / \*\$OMP directive [clause[,] clause]... (free format)
- Compiling OpenMP source code :
  - (**GNU Fortran compiler**) : `gfortran -fopenmp -o exec_name file_name.f95`
  - (**Intel Fortran compiler**) : `ifort -o exe_file_name -openmp file_name.f`

# DEMO: Hello World

```
[LSU760000@n00 l12]$ ls
hello.c reduction.c vector1.c vectorSections.c
[LSU760000@n00 l12]$ gcc -fopenmp -o hello hello.c
[LSU760000@n00 l12]$ ls
hello hello.c reduction.c vector1.c vectorSections.c
[LSU760000@n00 l12]$ export OMP_NUM_THREADS=4
[LSU760000@n00 l12]$ ./hello
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 3
```

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- **OpenMP: Data & Work sharing directives**
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



# OpenMP: Data Environment

- OpenMP program always begins with a single thread of control – master thread
- Context associated with the master thread is also known as the Data Environment.
- Context is comprised of :
  - Global variables
  - Automatic variables
  - Dynamically allocated variables
- Context of the master thread remains valid throughout the execution of the program
- The OpenMP parallel construct may be used to either share a single copy of the context with all the threads or provide each of the threads with a private copy of the context.
- The sharing of Context can be performed at various levels of granularity
  - Select variables from a context can be shared while keeping the context private etc.

# OpenMP Data Environment

- OpenMP data scoping clauses allow a programmer to decide a variable's execution context (should a variable be shared or private.)
- 3 main data scoping clauses in OpenMP (Shared, Private, Reduction) :
- Shared :
  - A variable will have a single storage location in memory for the duration of the parallel construct, i.e. references to a variable by different threads access the same memory location.
  - That part of the memory is shared among the threads involved, hence modifications to the variable can be made using simple read/write operations
  - Modifications to the variable by different threads is managed by underlying shared memory mechanisms
- Private :
  - A variable will have a separate storage location in memory for each of the threads involved for the duration of the parallel construct.
  - All read/write operations by the thread will affect the thread's private copy of the variable .
- Reduction :
  - Exhibit both shared and private storage behavior. Usually used on objects that are the target of arithmetic reduction.
  - Example : summation of local variables at the end of a parallel construct

# OpenMP Work-Sharing Directives

- Work sharing constructs divide the execution of the enclosed block of code among the group of threads.
- They do not launch new threads.
- No implied barrier on entry
- Implicit barrier at the end of work-sharing construct
- Commonly used Work Sharing constructs :
  - **for** directive (C/C++ ; equivalent DO construct available in Fortran but will not be covered here) : shares iterations of a loop across a group of threads
  - **sections** directive : breaks work into separate sections between the group of threads; such that each thread independently executes a section of the work.
  - **critical** directive: serializes a section of code

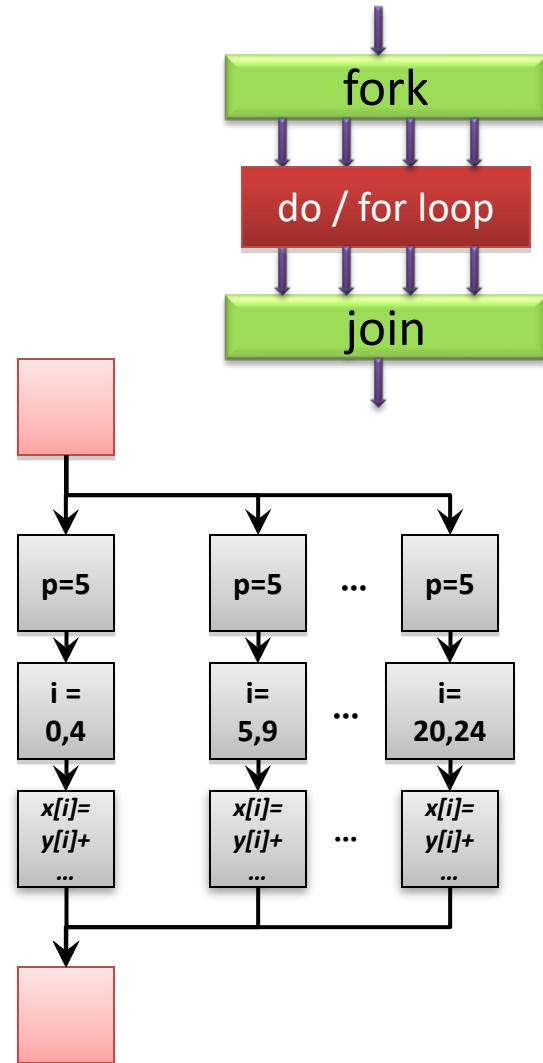
# OpenMP Schedule Clause

- The schedule clause defines how the iterations of a loop are divided among a group of threads
- **static** : iterations are divided into pieces of size chunk and are statically assigned to each of the threads in a round robin fashion
- **dynamic** : iterations divided into pieces of size chunk and dynamically assigned to a group of threads. After a thread finishes processing a chunk, it is dynamically assigned the next set of iterations.
- **guided** : For a chunk of size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk with value k, the same algorithm is used for determining the chunk size with the constraint that no chunk should have less than k chunks except the last chunk.
- Default schedule is implementation specific while the default chunk size is usually 1

# OpenMP for directive

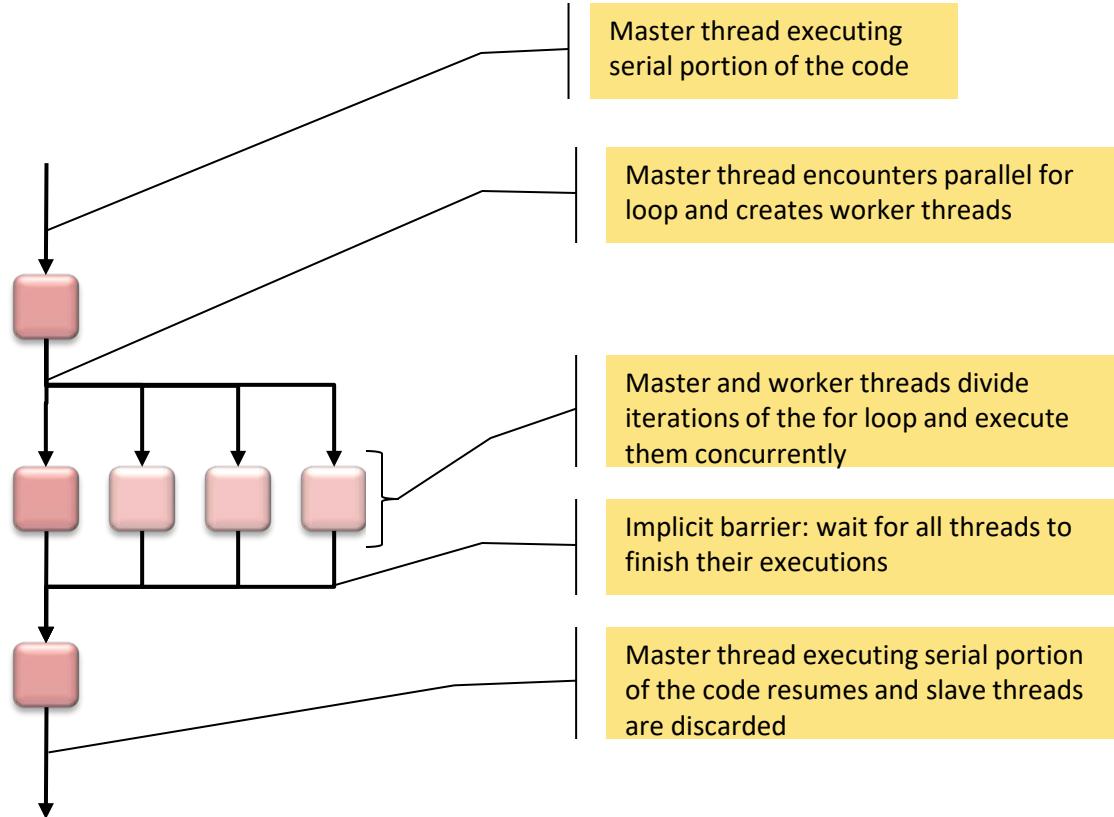
- `for` directive helps share iterations of a loop between a group of threads
- If `nowait` is specified then the threads do not wait for synchronization at the end of a parallel loop
- The `schedule` clause describes how iterations of a loop are divided among the threads in the team (discussed in detail in the next few slides)

```
#pragma omp parallel
{
    p=5;
    #pragma omp for
        for (i=0; i<24; i++)
            x[i]=y[i]+p*(i+3)
        ...
    ...
} /* omp end parallel */
```



# Simple Loop Parallelization

```
#pragma omp parallel for  
for (i=0; i<n; i++)  
    z( i ) = a*x(i)+y
```



# Example: OpenMP work sharing Constructs

```
#include <omp.h>
#define N    16
main ()
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = 4;
printf("a[i] + b[i] = c[i] \n");
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < N; i++)
  c[i] = a[i] + b[i];
} /* end of parallel section */
for (i=0; i < N; i++)
  printf(" %f + %f = %f \n",a[i],b[i],c[i]);
}
```

Initializing the vectors a[i], b[i]

Instructing the runtime environment that a,b,c,chunk are shared variables and I is a private variable

The nowait ensures that the child threads do not synchronize once their work is completed

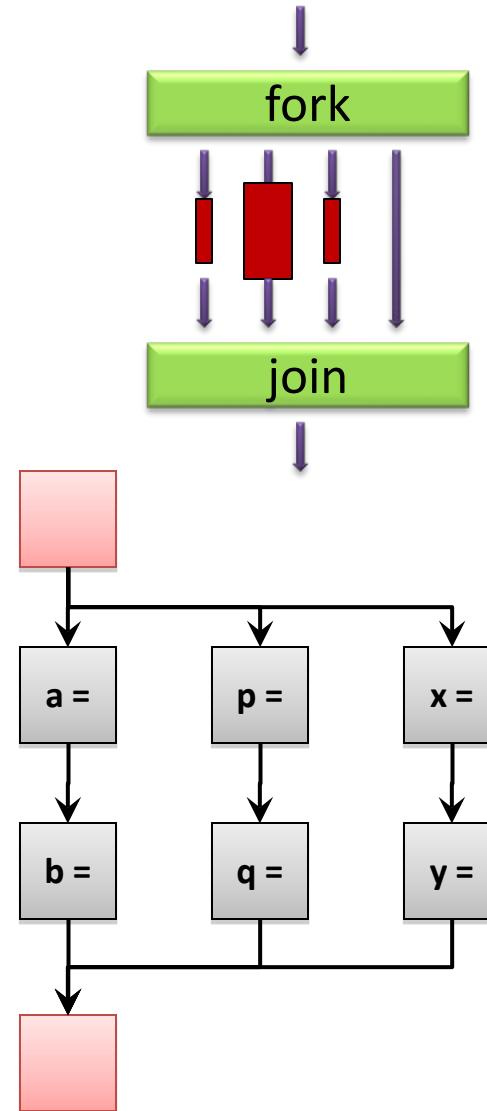
Load balancing the threads using a DYNAMIC policy where array is divided into chunks of 4 and assigned to the threads

Modified from examples posted on:  
<https://computing.llnl.gov/tutorials/openMP/>

# OpenMP sections directive

- `sections` directive is a non iterative work sharing construct.
- Independent `section` of code are nested within a `sections` directive
- It specifies enclosed `section` of codes between different threads
- Code enclosed within a `section` directive is executed by a thread within the pool of threads

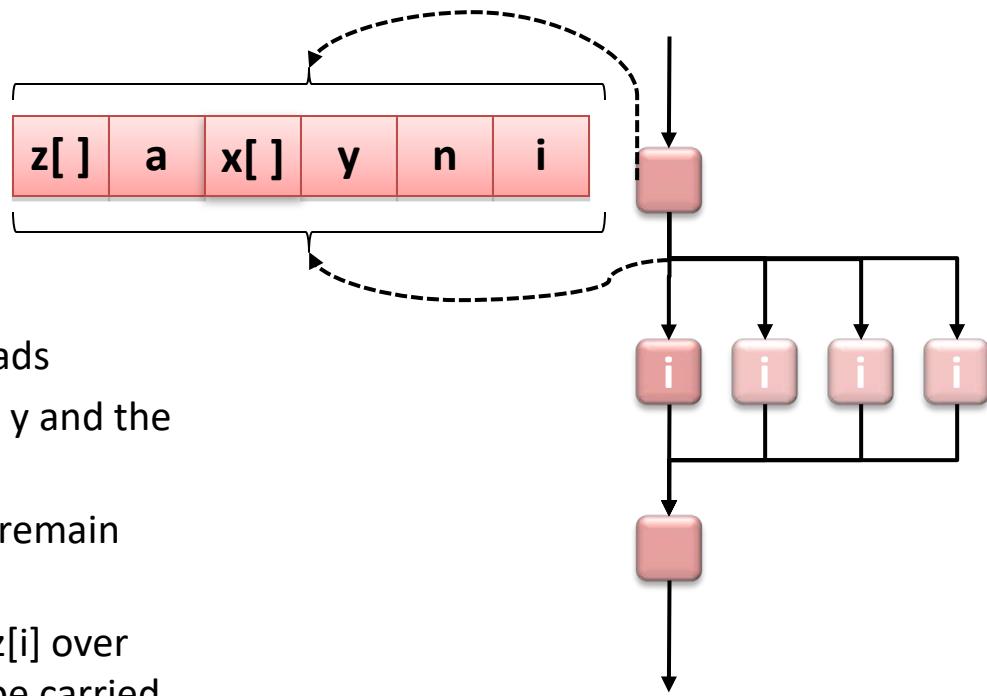
```
#pragma omp parallel private(p)
{
    #pragma omp sections
    {{  a=...;
        b=...; }
        #pragma omp section
        {  p=...;
            q=...; }
        #pragma omp section
        {  x=...;
            y=...; }
    } /* omp end sections */
} /* omp end parallel */
```



# Understanding variables in OpenMP

```
#pragma omp parallel for
for (i=0; i<n; i++)
    z[i] = a*x[i]+y
```

- Shared variable `z` is modified by multiple threads
- Each iteration reads the scalar variables `a` and `y` and the array element `x[i]`
- `a,y,x` can be read concurrently as their values remain unchanged.
- Each iteration writes to a distinct element of `z[i]` over the index range. Hence write operations can be carried out concurrently with each iteration writing to a distinct array index and memory location
- The parallel `for` directive in OpenMP ensures that the `for` loop index value (`i` in this case) is private to each thread.



# Example: OpenMP Sections

```
#include <omp.h>
#define N 16
main (){
    int i;
    float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.5;
#pragma omp parallel shared(a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];
    } /* end of sections */
} /* end of parallel section */
...
```

Sections construct that encloses the section calls

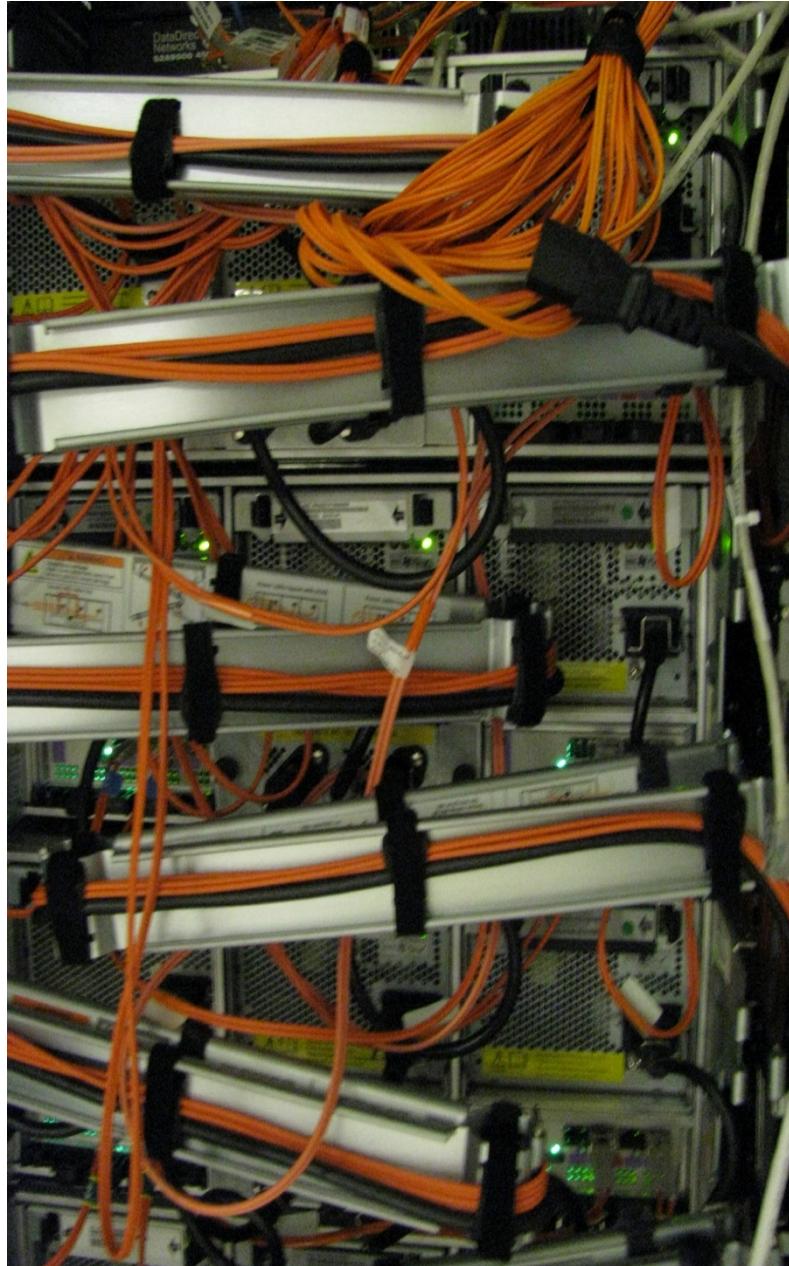
Section : that computes the sum of the 2 vectors

Section : that computes the product of the 2 vectors

Modified from examples posted on:  
<https://computing.llnl.gov/tutorials/openMP/>

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- **OpenMP: Synchronization**
- OpenMP: Reduction
- Synopsis of Commands



# Thread Synchronization

- “communication” mainly through read write operations on shared variables
- Synchronization defines the mechanisms that help in coordinating execution of multiple threads (that use a shared context) in a parallel program.
- Without synchronization, multiple threads accessing shared memory location may cause conflicts by :
  - Simultaneously attempting to modify the same location
  - One thread attempting to read a memory location while another thread is updating the same location.
- Synchronization helps by providing explicit coordination between multiple threads.
- Two main forms of synchronization :
  - Implicit event synchronization
  - Explicit synchronization – critical, master directives in OpenMP

# Basic Types of Synchronization

- Explicit Synchronization via mutual exclusion
  - Controls access to the shared variable by providing a thread exclusive access to the memory location for the duration of its construct.
  - Critical directive of OpenMP provides mutual exclusion
- Event Synchronization
  - Signals occurrence of an event across multiple threads.
  - Barrier directives in OpenMP provide the simplest form of event synchronization
  - The barrier directive defines a point in a parallel program where each thread waits for all other threads to arrive. This helps to ensure that all threads have executed the same code in parallel upto the barrier.
  - Once all threads arrive at the point, the threads can continue execution past the barrier.
- Additional synchronization mechanisms available in OpenMP

# OpenMP Synchronization: master

- The **master** directive in OpenMP marks a block of code that gets executed on a single thread.
- The rest of the threads in the group ignore the portion of code marked by the master directive
- Example

```
#pragma omp master  
structured block
```

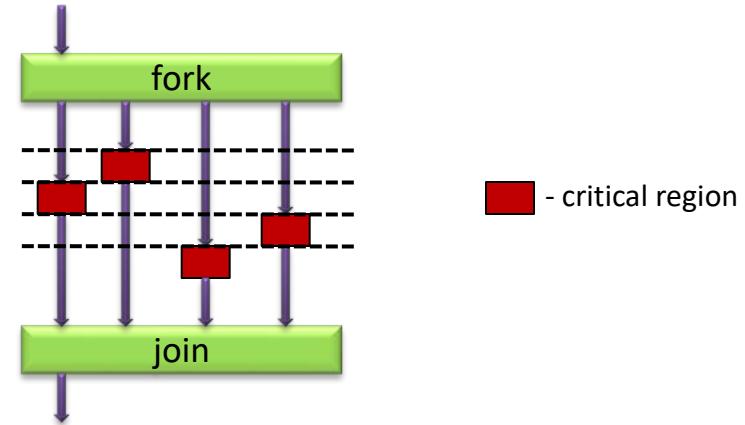
## Race Condition :

Two asynchronous threads access the same shared variable and at least one modifies the variable and the sequence of operations is undefined . Result of these asynchronous operations depends on detailed timing of the individual threads of the group.

# OpenMP critical directive: Explicit Synchronization

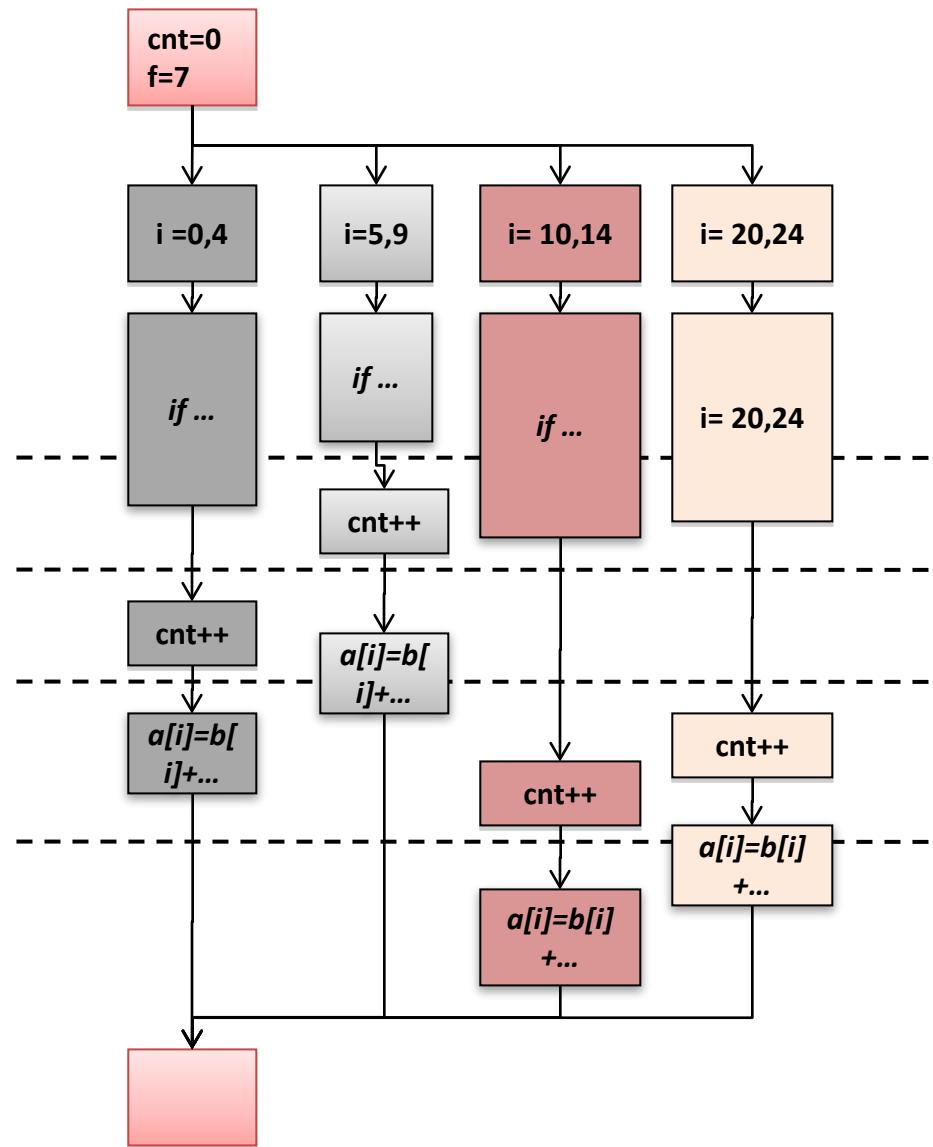
- Race conditions can be avoided by controlling access to shared variables by allowing threads to have exclusive access to the variables
- Exclusive access to shared variables allows the thread to atomically perform read, modify and update operations on the variable.
- Mutual exclusion synchronization is provided by the `critical` directive of OpenMP
- Code block within the `critical region` defined by `critical /end critical` directives can be executed only by one thread at a time.
- Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

```
int x
x=0;
#pragma omp parallel shared(x)
{
    #pragma omp critical
        x = 2*x + 1;
} /* omp end parallel */
```



# Simple Example: critical

```
cnt = 0;  
f = 7;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0;i<20;i++){  
        if(b[i] == 0){  
  
            #pragma omp critical  
            cnt ++;  
        } /* end if */  
        a[i]=b[i]+f*(i+1);  
    } /* end for */  
} /* omp end parallel */
```



# Topics

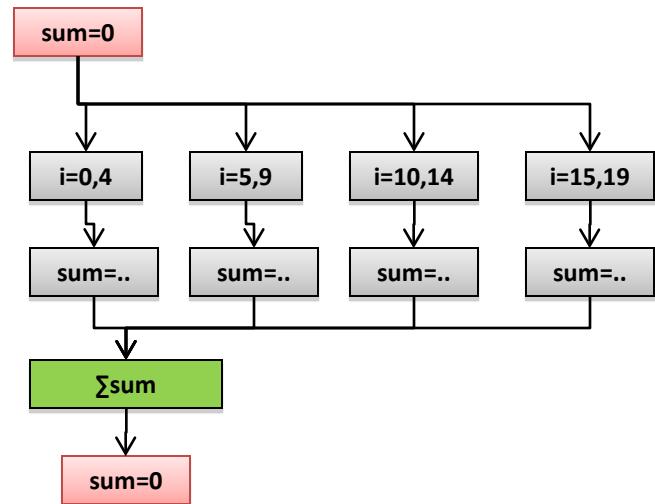
- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- **OpenMP: Reduction**
- Synopsis of Commands



# OpenMP: Reduction

- performs reduction on *shared variables* in list based on the *operator* provided.
- for C/C++ operator can be any one of :
  - +, \*, -, ^, |, ||, & or &&
  - At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```



# Example: Reduction

```
#include <omp.h>
main () {
int i, n, chunk;
float a[16], b[16], result;
n = 16;
chunk = 4;
result = 0.0;
for (i=0; i < n; i++)
{
    a[i] = i * 1.0;
    b[i] = i * 2.0;
}
```

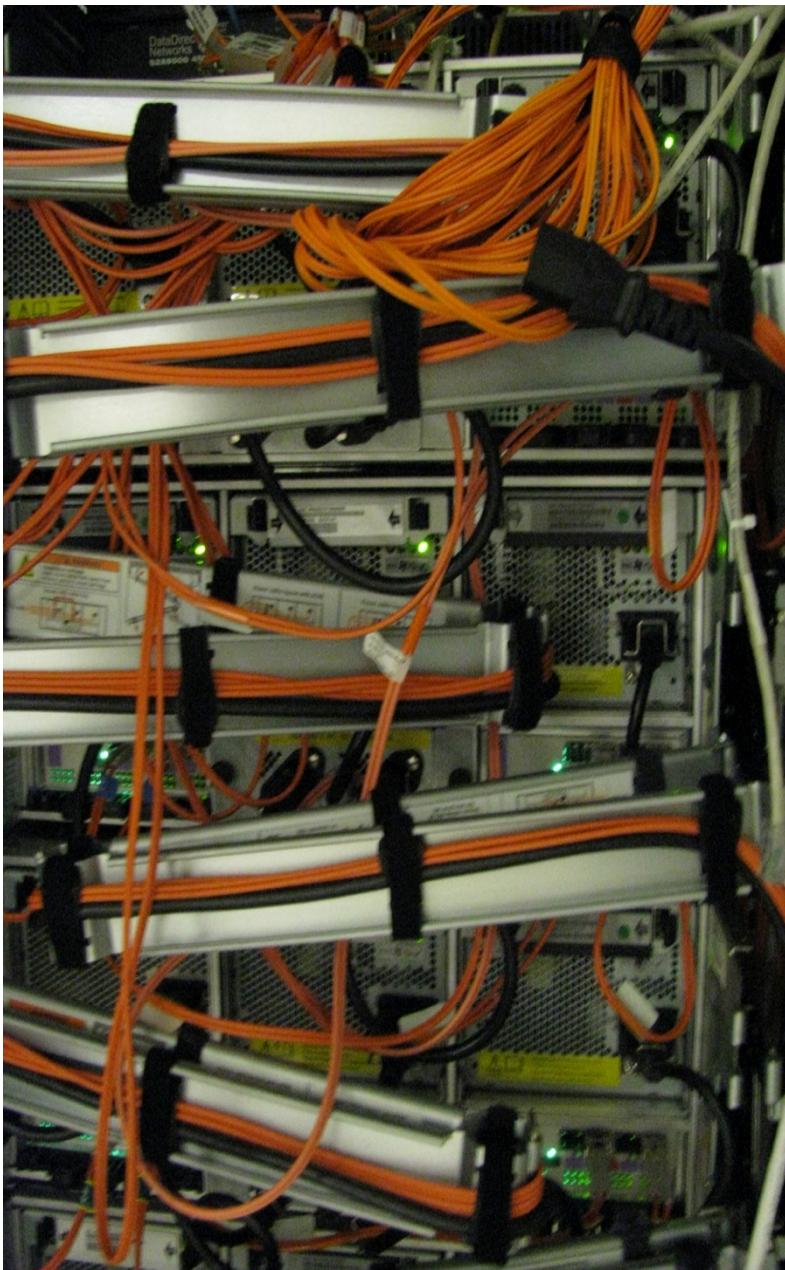
Reduction example with summation where the result of the reduction operation stores the dotproduct of two vectors  
 $\sum a[i]*b[i]$

```
#pragma omp parallel for default(shared) private(i) \
schedule(static,chunk) reduction(+:result)
for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```

SRC : <https://computing.llnl.gov/tutorials/openMP/>

# Topics

- Review of HPC Models
- Shared Memory: Performance concepts
- Introduction to OpenMP
- OpenMP: Runtime Library & Environment Variables
- OpenMP: Data & Work sharing directives
- OpenMP: Synchronization
- OpenMP: Reduction
- Synopsis of Commands



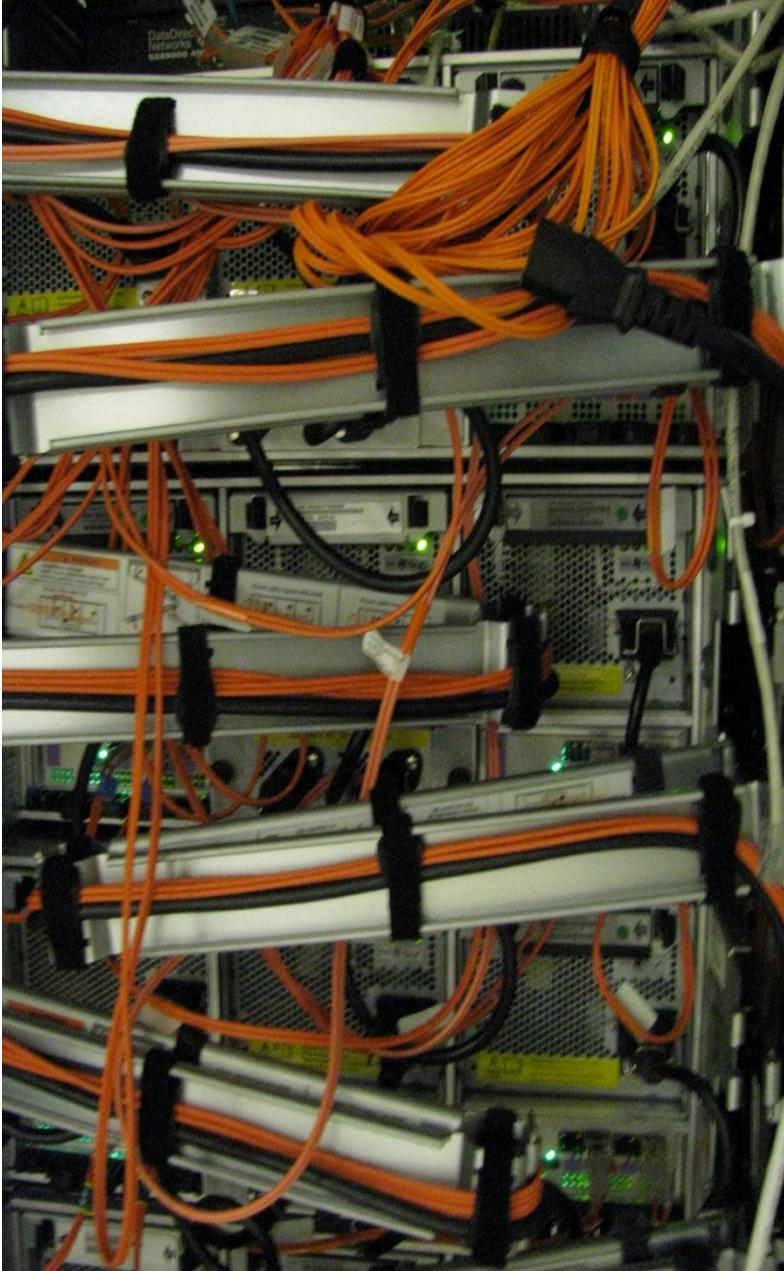
# Synopsis of Commands

- How to invoke OpenMP runtime systems `#pragma omp parallel`
- The interplay between OpenMP environment variables and runtime system (`omp_get_num_threads()`,  
`omp_get_thread_num()`)
- Shared data directives such as `shared`, `private` and `reduction`
- Basic flow control using `sections`, `for`
- Fundamentals of synchronization using `critical` directive and critical section.
- And directives used for the OpenMP programming part of the problem set.

# The Essential MPI

# Message Passing Interface MPI

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives



# Opening Remarks

- Context: distributed memory parallel computers
- We have communicating sequential processes, each with their own memory, and no access to another process's memory
  - A fairly common scenario from the mid 1980s (Intel Hypercube) to today
  - Processes interact (exchange data, synchronize) through message passing
  - Initially, each computer vendor had its own library and calls
  - First standardization was PVM
    - Started in 1989, first public release in 1991
    - Worked well on distributed machines
    - Next was MPI

# What You' II Need to Know

- What is a standard API
- How to build and run an MPI-1 program
- Basic MPI functions
  - 4 basic environment functions
    - Including the idea of communicators
  - Basic point-to-point functions
    - Blocking and non-blocking
    - Deadlock and how to avoid it
    - Data types
  - Basic collective functions
- The advanced MPI-1 material may be required for the problem set
- The MPI-2 highlights are just for information

# MPI Standard

- From 1992-1994, a community representing both vendors and users decided to create a standard interface to message passing calls in the context of distributed memory parallel computers (MPPs, there weren't really clusters yet)
- MPI-1 was the result
  - “Just” an API
  - FORTRAN77 and C bindings
  - Reference implementation (mpich) also developed
  - Vendors also kept their own internals (behind the API)

# MPI Standard

- Since then
  - MPI-1.1
    - Fixed bugs, clarified issues
  - MPI-2
    - Included MPI-1.2
      - Fixed more bugs, clarified more issues
    - Extended MPI
      - New datatype constructors, language interoperability
    - New functionality
      - One-sided communication
      - MPI I/O
      - Dynamic processes
    - FORTRAN90 and C++ bindings
- Best MPI reference
  - MPI Standard - on-line at: <http://www mpi-forum.org/>

# MPI: Basics

- Every MPI program must contain the preprocessor directive

```
#include "mpi.h"
```

- The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.
- mpi.h is usually found in the “include” directory of most MPI installations. For example on arete:

The screenshot shows a terminal window with the following content:

```
[lsu00@master ~]$ ls /usr/include/mpich2-x86_64/
clog_commset.h    mpe_graphicsf.h   mpe_log_thread.h   mpif.h      mpi_sizeofs.mod  primitives
clog_const.h      mpe_graphics.h    mpe_misc.h       mpi.h      opa_config.h
clog_inttypes.h   mpe.h           mpe_base.mod     mpi.mod    opa_primitives.h
clog_uuid.h       mpe_logf.h      mpe_constants.mod  mpiof.h   opa_queue.h
mpe_callstack.h   mpe_log.h      mpicxx.h        mpio.h    opa_util.h
[lsu00@master ~]$
```

Below the terminal window, a portion of an MPI C code is shown, enclosed in a red box:

```
...
#include "mpi.h"

...
MPI_Init(&Argc,&Argv);
...
...
MPI_Finalize();
...
```

# MPI: Initializing MPI Environment

---

Function:      **MPI\_init()**

```
int MPI_Init(int *argc, char ***argv)
```

Description:

Initializes the MPI execution environment. **MPI\_init()** must be called before any other MPI functions can be called and it should be called only once. It allows systems to do any special setup so that MPI Library can be used. **argc** is a pointer to the number of arguments and **argv** is a pointer to the argument vector. On exit from this routine, all processes will have a copy of the argument list.

---

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

# MPI: Terminating MPI Environment

---

Function:      **MPI\_Finalize()**

```
int MPI_Finalize()
```

Description:

Terminates MPI execution environment. All MPI processes must call this routine before exiting. **MPI\_Finalize()** need not be the last executable statement or even in main; it must be called at somepoint following the last call to any other MPI function.

---

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize( );
...
```

# MPI Hello World

- C source file for a simple MPI Hello World

```
#include "mpi.h"  
#include <stdio.h>
```

Include header files

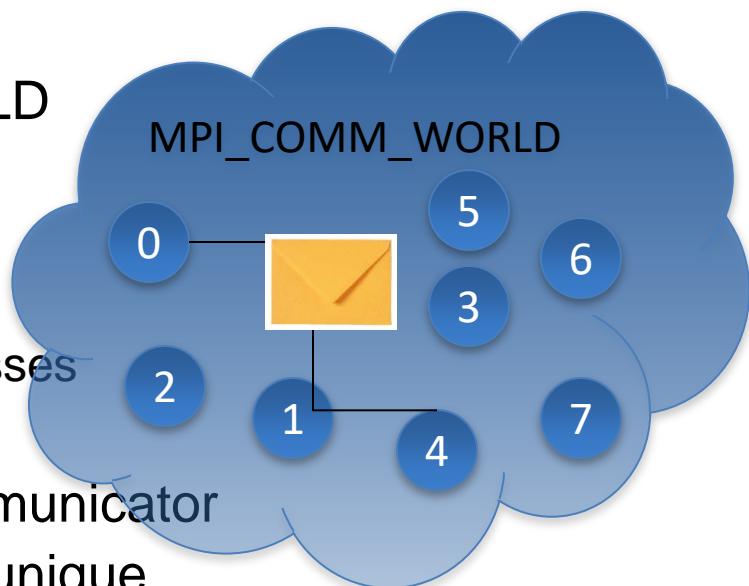
```
int main( int argc, char *argv[] )  
{  
    MPI_Init( &argc, &argv );  
    printf("Hello, World! \n");  
    MPI_Finalize();  
    return 0;  
}
```

Initialize MPI Context

Finalize MPI Context

# MPI Communicators

- Communicator is an internal object
- MPI Programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)
- MPI provides functions to interact with it
- Default communicator is MPI\_COMM\_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - One can think of it as an ordered list of processes
- Additional communicator(s) can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank



# MPI: Size of Communicator

---

Function:      **MPI\_Comm\_size()**

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

Description:

Determines the size of the group associated with a communicator (*comm*). Returns an integer number of processes in the group underlying *comm* executing the program. If *comm* is an inter-communicator (i.e. an object that has processes of two inter-communicating groups), return the size of the local group (a size of a group where request is initiated from). The *comm* in the argument list refers to the communicator-group to be queried, the result of the query (size of the *comm* group) is stored in the variable *size*.

```
...
#include "mpi.h"
...
int size;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

# MPI: Rank of a process in comm

---

Function: **MPI\_Comm\_rank()**

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Description:

Returns the rank of the calling process in the group underlying the *comm*. If the *comm* is an inter-communicator, the call `MPI_Comm_rank` returns the rank of the process in the local group. The first parameter *comm* in the argument list is the communicator to be queried, and the second parameter *rank* is the integer number rank of the process in the group of *comm*.

---

```
...
#include "mpi.h"
...
int rank;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

# Example: communicators

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("Hello, World! from %d of %d\n", rank,
size );
    MPI_Finalize();
    return 0;
}
```

Determines the rank of the current process in the communicator-group  
MPI\_COMM\_WORLD

Determines the size of the communicator-group  
MPI\_COMM\_WORLD

...  
Hello, World! from 1 of 8  
Hello, World! from 0 of 8  
Hello, World! from 5 of 8  
...

# Example: Communicator & Rank

- Compiling :

```
mpicc -o hello2 hello2.c
```

- Result :

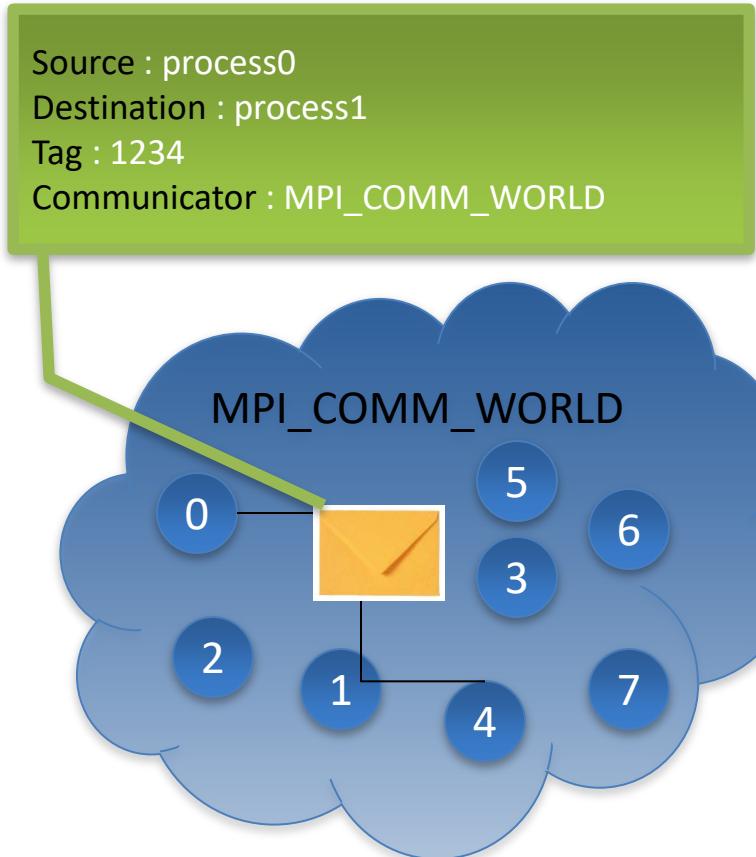
```
Hello, World! from 4 of 8
Hello, World! from 3 of 8
Hello, World! from 1 of 8
Hello, World! from 0 of 8
Hello, World! from 5 of 8
Hello, World! from 6 of 8
Hello, World! from 7 of 8
Hello, World! from 2 of 8
```

# MPI : Point to Point Communication primitives

- A basic communication mechanism of MPI between a pair of processes in which one process is sending data and the other process receiving the data, is called “*point to point communication*”
- Message passing in MPI program is carried out by 2 main MPI functions
  - MPI\_Send – sends message to a designated process
  - MPI\_Recv – receives a message from a process
- Each of the *send* and *recv* calls is appended with additional information along with the data that needs to be exchanged between application programs
- The message envelope consists of the following information
  - The rank of the receiver
  - The rank of the sender
  - A tag
  - A communicator
- The source argument is used to distinguish messages received from different processes
- Tag is user-specified *int* that can be used to distinguish messages from a single process

# Message Envelope

- Communication across processes is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - Envelope comprises **source**, **destination**, **tag**, **communicator**
  - Message = Envelope + Data
- Communicator refers to the namespace associated with the group of related processes



# MPI: (blocking) Send message

Function: **MPI\_Send()**

```
int MPI_Send(  
            void          *message,  
            int           count,  
            MPI_Datatype datatype,  
            int           dest,  
            int           tag,  
            MPI_Comm     comm )
```

## Description:

The contents of *message* are stored in a block of memory referenced by the first parameter *message*. The next two parameters, *count* and *datatype*, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype*. *MPI* allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage an overflow error occurs. The *dest* parameter corresponds to the rank of the process to which message has to be sent.

# MPI: Data Types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs

# MPI: (blocking) Receive message

Function: [MPI\\_Recv\(\)](#)

```
int MPI_Recv(  
            void          *message,  
            int           count,  
            MPI_Datatype datatype,  
            int           source,  
            int           tag,  
            MPI_Comm     comm,  
            MPI_Status   *status )
```

## Description:

The contents of message are stored in a block of memory referenced by the first parameter *message*. The next two parameters, *count* and *datatype*, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype*. *MPI* allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage an overflow error occurs. The *source* parameter corresponds to the rank of the process from which the message has been received. The *MPI\_Status* parameter in the *MPI\_Recv()* call returns information on the data that was actually received. It references a record with 2 fields – one for the source and one for the tag.

# MPI\_Status object

Object: **MPI\_Status**

Example usage :

```
MPI_Status status;
```

Description:

The MPI\_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI\_SOURCE), the message tag (MPI\_TAG), and error status (MPI\_ERROR) .

```
#include "mpi.h"
...
MPI_Status status; /* return status for */
...
MPI_Init(&argc, &argv);
...
if (my_rank != 0) {
...
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
    }
}
MPI_Finalize();
...
```

# MPI: Example send/recv

```
/* hello world, MPI style */

#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
    int my_rank;      /* rank of process */
    int p;           /* number of processes */
    int source;      /* rank of sender */
    int dest;        /* rank of receiver */

    int tag=0;        /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for */
                       /* receive */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that \0 gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
    }
    else { /* my rank == 0 */
        for (source = 1; source < p; source++ ) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
        printf("Greetings from process %d!\n", my_rank);
    }

    /* Shut down MPI */
    MPI_Finalize();

} /* end main */
```

# Communication map for the example.

`mpiexec -n 8 ./hello3`

*Greetings from process 1!*

*Greetings from process 2!*

*Greetings from process 3!*

*Greetings from process 4!*

*Greetings from process 5!*

*Greetings from process 6!*

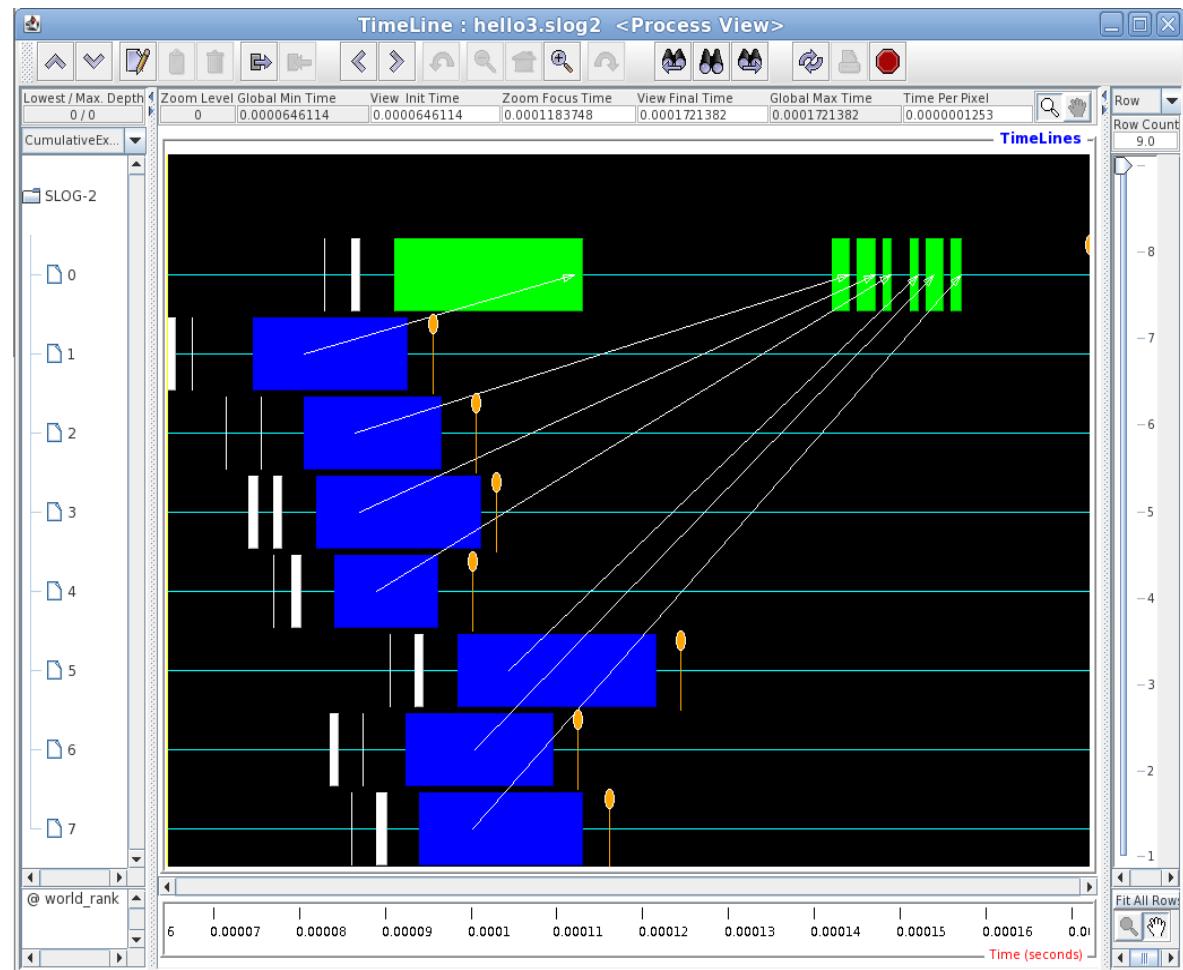
*Greetings from process 7!*

*Greetings from process 0!*

*Writing logfile....*

*Finished writing logfile.*

`[cdeka@celeritas i7]$`





# Review of Basic MPI Calls

- In review, the 6 main MPI calls:
  - MPI\_Init
  - MPI\_Finalize
  - MPI\_Comm\_size
  - MPI\_Comm\_rank
  - MPI\_Send
  - MPI\_Recv
- Include MPI Header file
  - #include “mpi.h”
- Basic MPI Datatypes
  - MPI\_INT, MPI\_FLOAT, ....

# Collective Calls

- A communication pattern that encompasses all processes within a communicator is known as **collective communication**
- MPI has several collective communication calls, the most frequently used are:
  - Synchronization
    - Barrier
  - Communication
    - Broadcast
    - Gather & Scatter
    - All Gather
  - Reduction
    - Reduce
    - AllReduce

# MPI Collective Calls: Barrier

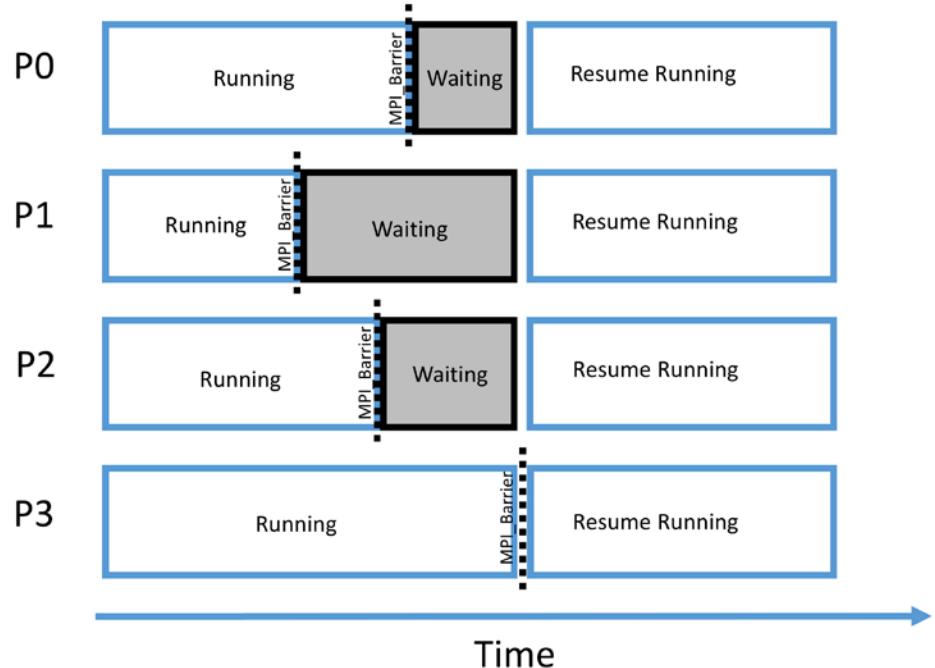
Function: **MPI\_Barrier()**

```
int MPI_Barrier (  
    MPI_Comm comm )
```

## Description:

Creates barrier synchronization in a communicator group *comm*. Each process, when reaching the MPI\_Barrier call, blocks until all the processes in the group reach the same MPI\_Barrier call.

[http://www-unix.mcs.anl.gov/mpi/www/www3/MPI\\_Barrier.html](http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Barrier.html)



# Example: MPI\_Barrier()

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[]){
    int      rank, size, len;
    char     name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
MPI_Barrier(MPI_COMM_WORLD);

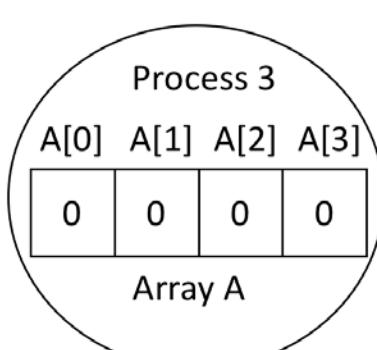
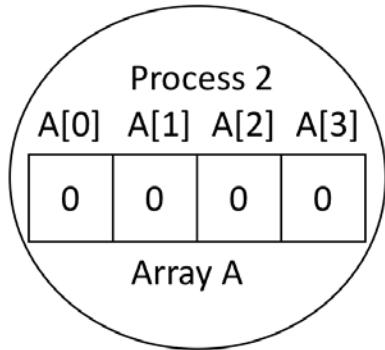
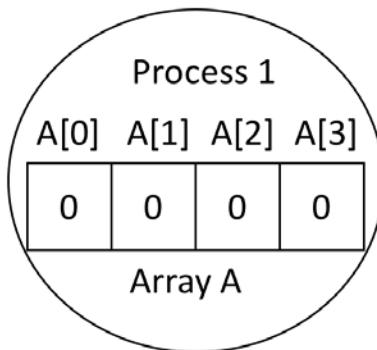
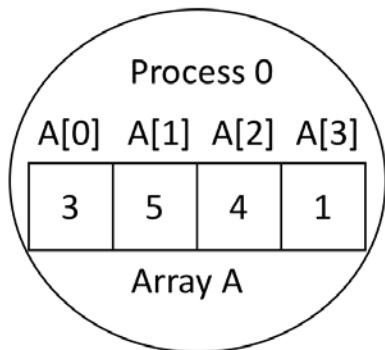
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &len);
MPI_Barrier(MPI_COMM_WORLD);

    printf ("Hello world! Process %d of %d on %s\n", rank,
size, name);
    MPI_Finalize();
    return 0;
}
```

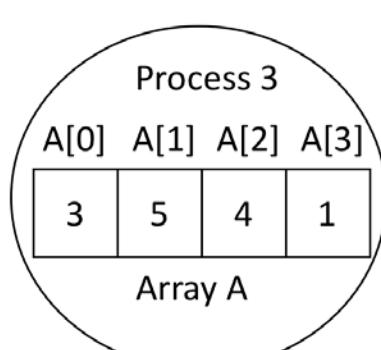
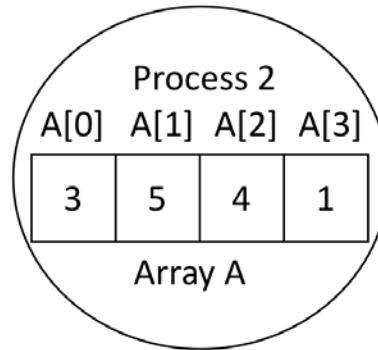
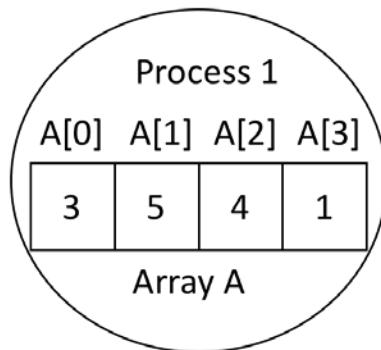
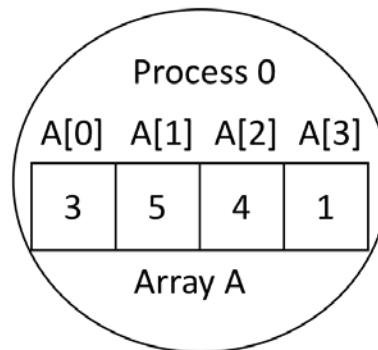
```
[cdekate@celeritas collective]$ mpirun -np 8 barrier
Hello world! Process 0 of 8 on celeritas.cct.lsu.edu
Writing logfile....
Finished writing logfile.
Hello world! Process 4 of 8 on compute-0-3.local
Hello world! Process 1 of 8 on compute-0-0.local
Hello world! Process 3 of 8 on compute-0-2.local
Hello world! Process 6 of 8 on compute-0-5.local
Hello world! Process 7 of 8 on compute-0-6.local
Hello world! Process 5 of 8 on compute-0-4.local
Hello world! Process 2 of 8 on compute-0-1.local
[cdekate@celeritas collective]$
```

# Collective Data Movement

Pre-Broadcast: Root Process 0

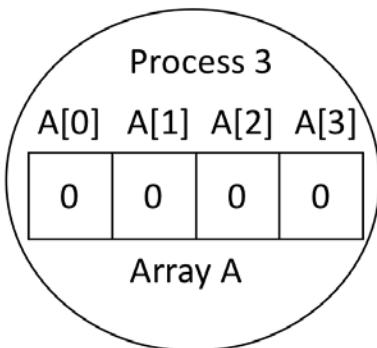
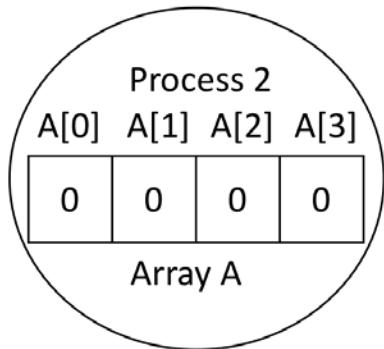
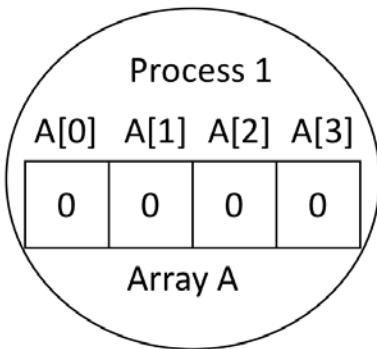
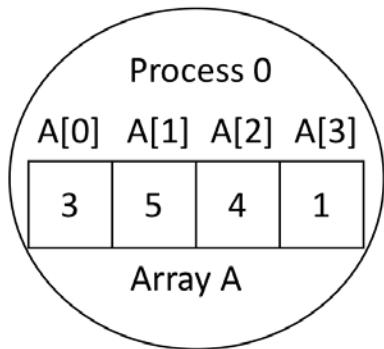


Post-Broadcast

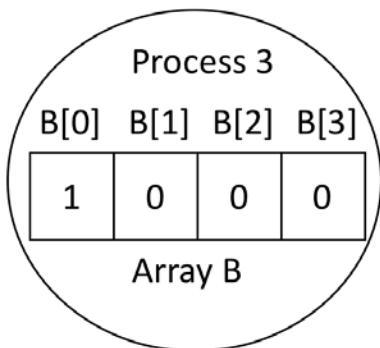
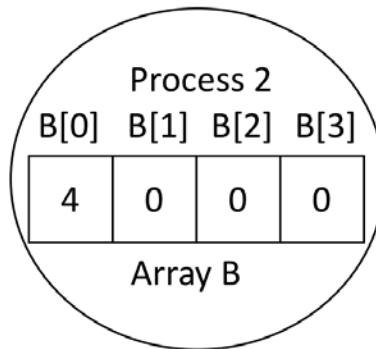
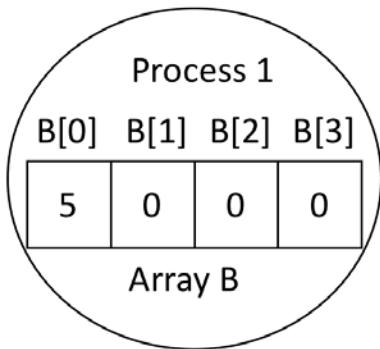
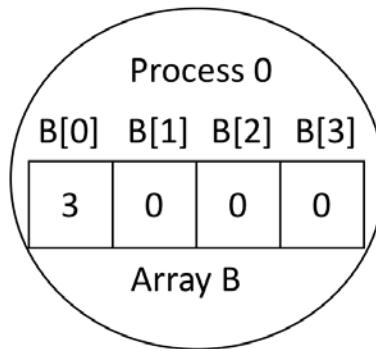


# Collective Data Movement

Pre-Scatter: Root process 0

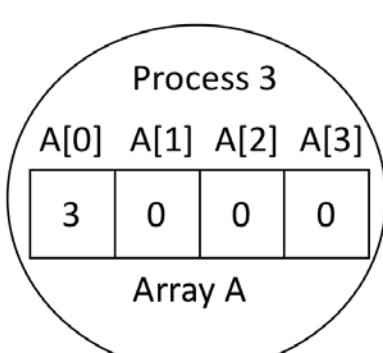
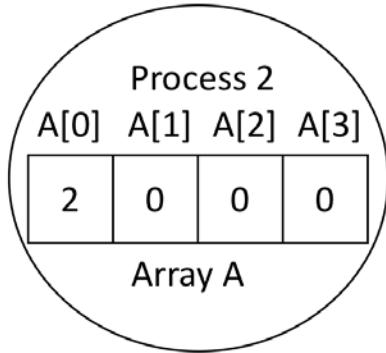
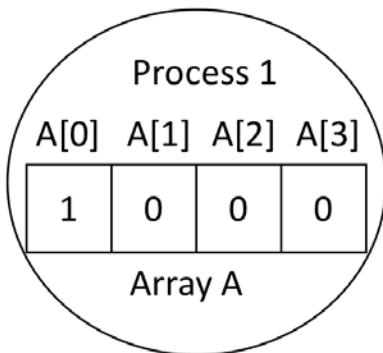
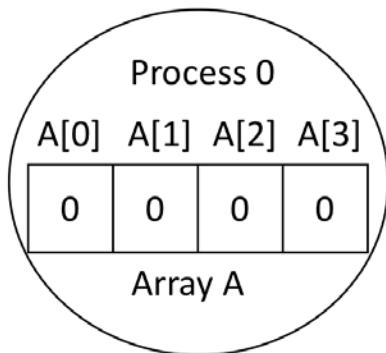


Post-Scatter

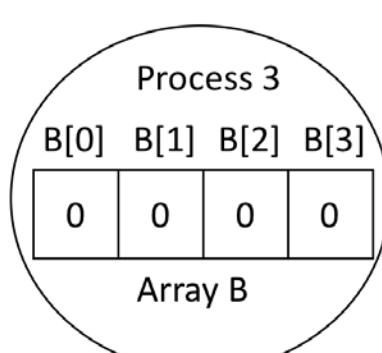
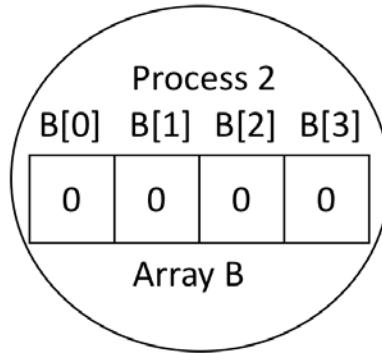
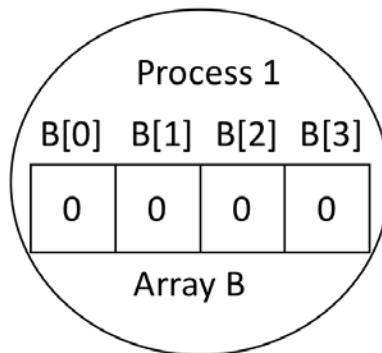
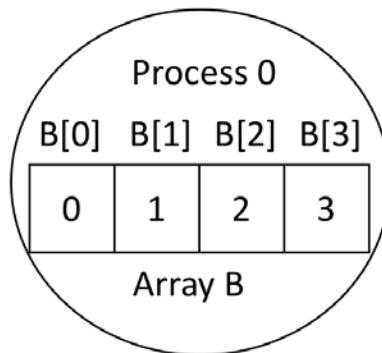


# Collective Data Movement

Pre-Gather: Root process 0

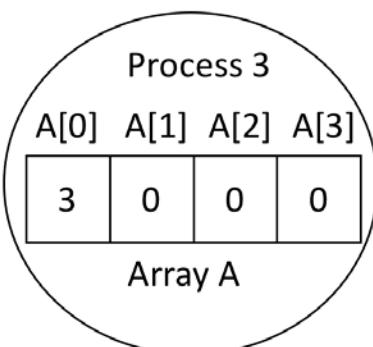
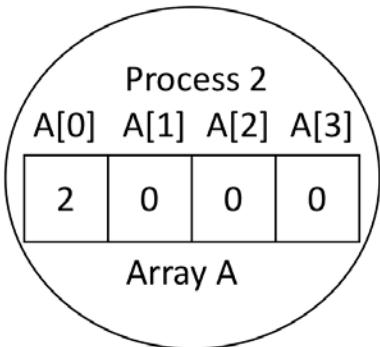
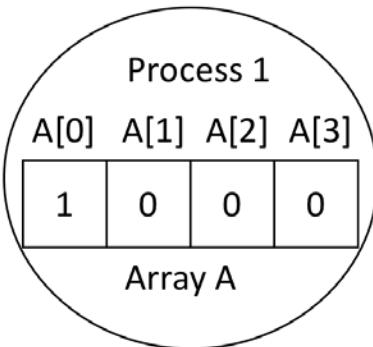
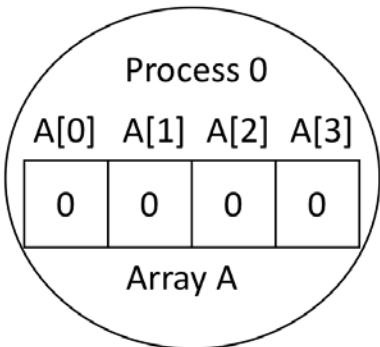


Post-Gather

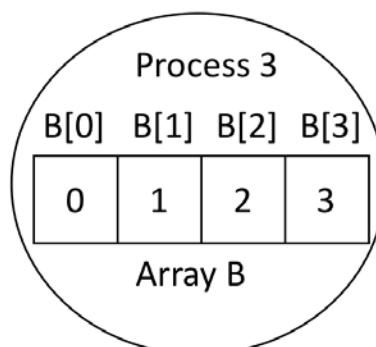
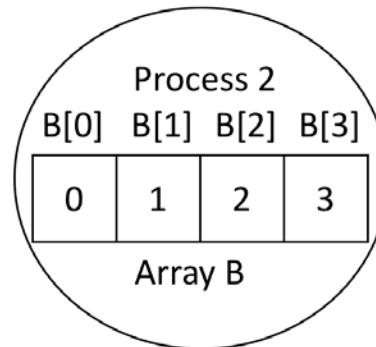
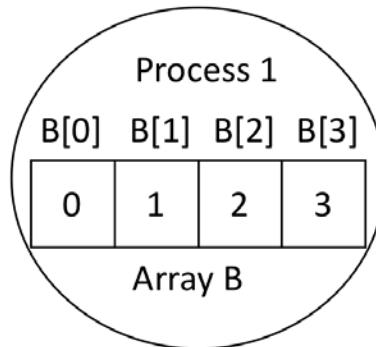
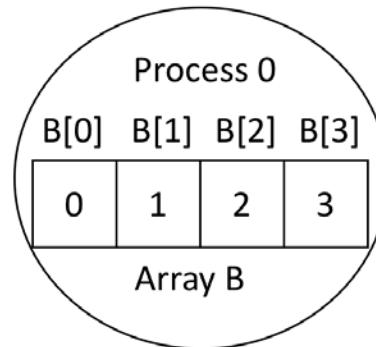


# Collective Data Movement

Pre-Allgather



Post-Allgather



# Collective Data Movement

Pre Alltoall

Process	Data partition			
	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

Post Alltoall

Process	Data partition			
	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

# MPI Collective Calls: Broadcast

Function: **MPI\_Bcast()**

```
int MPI_Bcast (  
    void          *message,  
    int           count,  
    MPI_Datatype datatype,  
    int           root,  
    MPI_Comm      comm )
```

Description:

A collective communication call where a single process sends the same data contained in the *message* to every process in the communicator. By default a tree like algorithm is used to broadcast the message to a block of processors, a linear algorithm is then used to broadcast the message from the first process in a block to all other processes. All the processes invoke the *MPI\_Bcast* call with the same arguments for *root* and *comm*,

```
float      endpoint[2];  
...  
MPI_Bcast(endpoint, 2, MPI_FLOAT, 0, MPI_COMM_WORLD);  
...
```

# MPI Collective Calls: Scatter

Function: [MPI\\_Scatter\(\)](#)

```
int MPI_Scatter (
    void          *sendbuf,
    int           send_count,
    MPI_Datatype send_type,
    void          *recvbuf,
    int           recv_count,
    MPI_Datatype recv_type,
    int           root,
    MPI_Comm      comm)
```

Description:

`MPI_Scatter` splits the data referenced by the `sendbuf` on the process with rank `root` into  $p$  segments each of which consists of `send_count` elements of type `send_type`. The first segment is sent to process0 and the second segment to process1. The send arguments are significant on the process with rank `root`.

```
...
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT, row_segment, n/p, MPI_FLOAT, 0,MPI_COMM_WORLD);
...
```

# MPI Collective Calls: Gather

Function: **MPI\_Gather()**

```
int MPI_Gather (  
    void          *sendbuf,  
    int           send_count,  
    MPI_Datatype  sendtype,  
    void          *recvbuf,  
    int           recvcount,  
    MPI_Datatype  recvtype,  
    int           root,  
    MPI_Comm      comm )
```

Description:

`MPI_Gather` collects the data referenced by `sendbuf` from each process in the communicator `comm`, and stores the data in process rank order on the process with rank `root` in the location referenced by `recvbuf`. The `recv` parameters are only significant.

```
...  
    MPI_Gather( local_x, n/p, MPI_FLOAT, global_x, n/p, MPI_FLOAT, 0, MPI_COMM_WORLD );  
...
```

# MPI Collective Calls: All Gather

Function: **MPI\_Allgather()**

```
int MPI_Allgather (
    void          *sendbuf,
    int           send_count,
    MPI_Datatype  sendtype,
    void          *recvbuf,
    int           recvcount,
    MPI_Datatype  recvtype,
    MPI_Comm      comm )
```

Description:

MPI\_Allgather gathers the content from the send buffer (*sendbuf*) on each process. The effect of this call is similar to executing MPI\_Gather() *p* times with a different process acting as the root.

```
for (root=0; root<p; root++)
    MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p, MPI_FLOAT, root, MPI_COMM_WORLD);
...

```

**CAN BE REPLACED WITH :**

```
MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT, MPI_COMM_WORLD);
```

# Collective Example

```
#include "mpi.h"

int main ( int argc, char **argv )
{
    int n, i, pool_size, my_rank;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == ROOT) {
        printf("Enter the number of intervals: ");
        scanf("%d", &n);
        if (n==0) n=100;
    }

    MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
```

# Collective Example, Continued

```
h      = 1.0 / (double) n;
sum = 0.0;
for (i = my_rank + 1; i <= n; i += pool_size) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + a*a);
}
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, ROOT,
           MPI_COMM_WORLD);

if (my_rank == ROOT) printf("\npi is approximately %.16f\n", pi);

MPI_Finalize();

return 0;
}
```

# MPI Collective Calls: Reduce

Function: **MPI\_Reduce()**

```
int MPI_Reduce (  
    void          *operand,  
    void          *result,  
    int           count,  
    MPI_Datatype datatype,  
    MPI_Op        operator,  
    int           root,  
    MPI_Comm      comm )
```

## Description:

A collective communication call where all the processes in a communicator contribute data that is combined using binary operations (MPI\_Op) such as addition, max, min, logical, and, etc. MPI\_Reduce combines the operands stored in the memory referenced by *operand* using the operation *operator* and stores the result in *\*result*. MPI\_Reduce is called by all the processes in the communicator *comm* and for each of the processes *count*, *datatype* *operator* and *root* remain the same.

```
...  
MPI_Reduce(&local_integral, &integral, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);  
...
```

# MPI Binary Operations

- MPI binary operators are used in the MPI\_Reduce function call as one of the parameters. MPI\_Reduce performs a global reduction operation (dictated by the MPI binary operator parameter) on the supplied operands.
- The predefined MPI Operators used are :

**Table 8.2 Predefined Reduction Operations in MPI and Supported Predefined MPI Data Types**

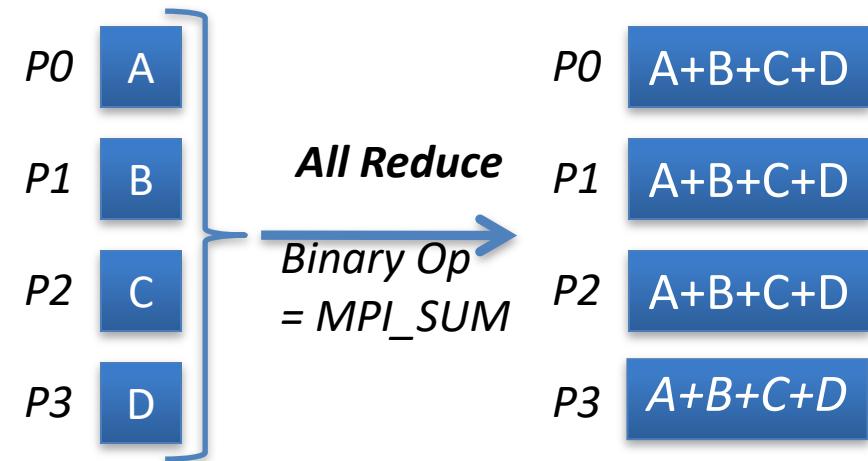
Predefined Reduction Operation	MPI Name	Supported Type
Maximum	MPI_MAX	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Minimum	MPI_MIN	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Summation	MPI_SUM	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Product	MPI_PROD	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Logical AND	MPI_LAND	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise AND	MPI_BAND	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Logical OR	MPI_LOR	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise OR	MPI_BOR	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Logical XOR	MPI_LXOR	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise XOR	MPI_BXOR	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Maximum value and location	MPI_MAXLOC	Pair data types: MPI_DOUBLE_INT (a double and an int), MPI_2INT (two ints)
Minimum value and location	MPI_MINLOC	Pair datatypes: MPI_DOUBLE_INT (a double and an int), MPI_2INT (two ints)

# MPI Collective Calls: All Reduce

Function: `MPI_Allreduce()`

```
int MPI_Allreduce (
```

```
    void          *sendbuf,  
    void          *recvbuf,  
    int           count,  
    MPI_Datatype datatype,  
    MPI_Op        op,  
    MPI_Comm      comm )
```



Description:

MPI\_Allreduce is used exactly like MPI\_Reduce, except that the result of the reduction is returned on all processes, as a result there is no *root* parameter.

```
...  
MPI_Allreduce(&integral, &integral, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);  
...
```

# Point to Point Communication Non-blocking Calls

- When the system buffer is full, the blocking send would have to wait until the receiving task pulled some message data out of the buffer. Use of non-blocking call allows computation to be done during this interval, allowing for interleaving of computation and communication
- **Non-blocking calls ensure that deadlock will not result**

# Deadlock

- Something to avoid
- A situation where the dependencies between processors are cyclic
  - One processor is waiting for a message from another processor, but that processor is waiting for a message from the first, so nothing happens
    - Until your time in the queue runs out and your job is killed
    - MPI does not have timeouts

# Deadlock Example

```
if (rank == 0) {  
    err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);  
    err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);  
} else {  
    err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);  
    err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);  
}
```

- If the message sizes are small enough, this should work because of systems buffers
- If the messages are too large, or system buffering is not used, this will hang

# Deadlock Example Solutions

```
If (rank == 0) {  
    err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);  
    err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);  
}else {  
    err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);  
    err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);  
}
```

or

```
If (rank == 0) {  
    err = MPI_Isend(sendbuf, count, datatype, 1, tag, comm, &req);  
    err = MPI_Irecv(recvbuf, count, datatype, 1, tag, comm);  
    err = MPI_Wait(req, &status);  
}else {  
    err = MPI_Isend(sendbuf, count, datatype, 0, tag, comm, &req);  
    err = MPI_Irecv(recvbuf, count, datatype, 0, tag, comm);  
    err = MPI_Wait(req, &status);  
}
```

# MPI Profiling: MPI\_Wtime

---

Function: **MPI\_Wtime()**

---

**double MPI\_Wtime()**

Description:

Returns time in seconds elapsed on the calling processor. Resolution of time scale is determined by the MPI environment variable MPI\_WTICK. When the MPI environment variable MPI\_WTIME\_IS\_GLOBAL is defined and set to true, the value of MPI\_Wtime is synchronized across all processes in MPI\_COMM\_WORLD

---

```
double time0;  
...  
time0 = MPI_Wtime();  
...  
printf("Hello From Worker #%-d %lf \n", rank, (MPI_Wtime() - time0));
```

[http://www-unix.mcs.anl.gov/mpi/www/www3/MPI\\_Wtime.html](http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Wtime.html)

# Timing Example: MPI\_Wtime

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
{
    int size, rank;
    double time0, time1;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    time0 = MPI_Wtime( );

    if(rank==0)
    {
        printf(" Hello From Proc0 Time = %lf \n", (MPI_Wtime( ) - time0) );
    }
    else
    {
        printf("Hello From Worker # %d %lf \n", rank, (MPI_Wtime( ) - time0) );
    }
    MPI_Finalize();
}
```

# Parallel Algorithms

# Parallel Algorithms

- Modern supercomputers employ several different modalities of operation to take advantage of parallelism
  - Three of these most common hardware architecture forms present in a supercomputer are
    - Single-Instruction Multiple Data (SIMD) parallelism,
    - shared memory parallelism,
    - and distributed memory parallelism.
- Shared memory parallelism and distributed memory parallelism are subclasses of the Multiple Instruction Multiple Data (MIMD) class of Flynn's computer architecture taxonomy.

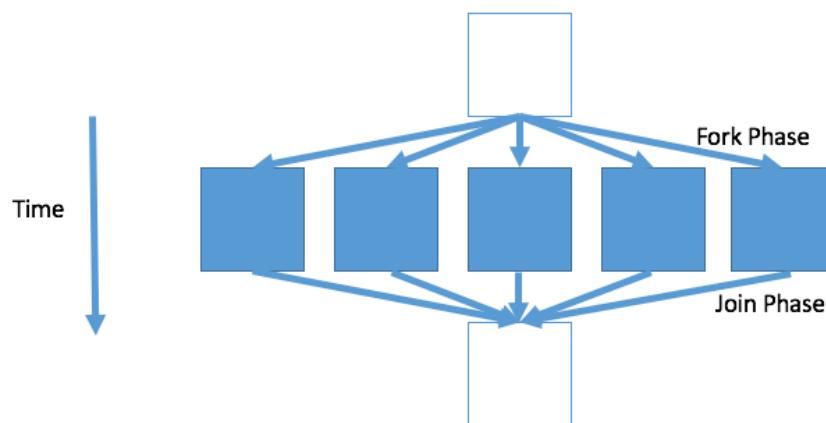
# Classes of Parallel Algorithms

- Among the many parallel algorithms used on modern supercomputers, several classes of parallel algorithms arise that share key characteristics and are driven by the underlying mechanism from which the parallelism is derived.

Generic Class of Parallel Algorithm	Example
Fork-Join	OpenMP parallel for-loop
Divide and Conquer	Fast Fourier Transform, Parallel Sort
Halo-Exchange	Finite difference/Finite Element partial differential equation solvers
Permutation	Cannon's algorithm, Fast Fourier Transform
Embarrassingly Parallel	Monte Carlo
Manager Worker	Simple Adaptive Mesh Refinement
Task dataflow	Breadth first search

# Fork-Join

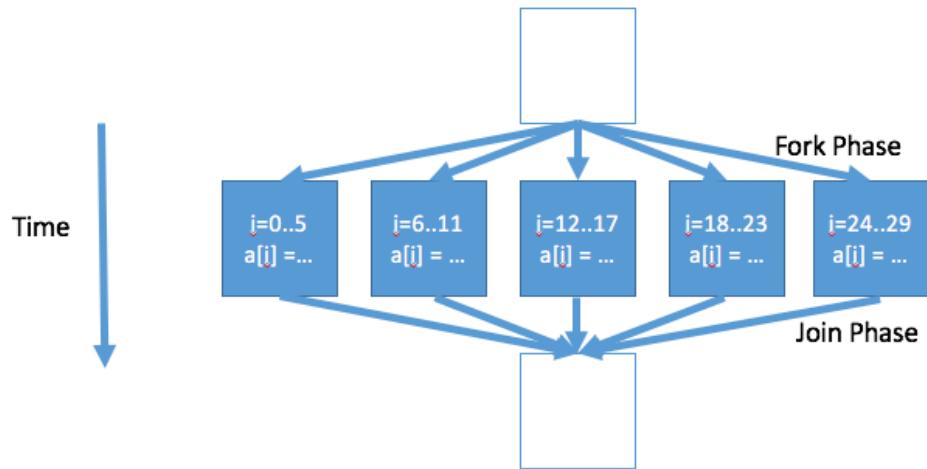
- The fork-join parallel design pattern is a key component of the OpenMP execution model
- is frequently employed in programming models targeting shared memory parallelism
- In regions of a sequential algorithm where work can proceed concurrently, a group of lightweight concurrent operators frequently called “threads” are created to perform that work
- Once the work is completed, the results from each of these operators are accumulated during the “join” phase



# Fork-Join Example

A previously initialized array  $b$  is added to another expression to initialize array  $a$ . Because each work element in the for-loop (see line 3) is independent of every other element, the work in this loop can proceed concurrently. Consequently, a parallel for-loop construct is added in line 1.

```
1 #pragma omp parallel for
2   for (i=0;i<30;i++)
3     a[i] = b[i] + sin(i)
```

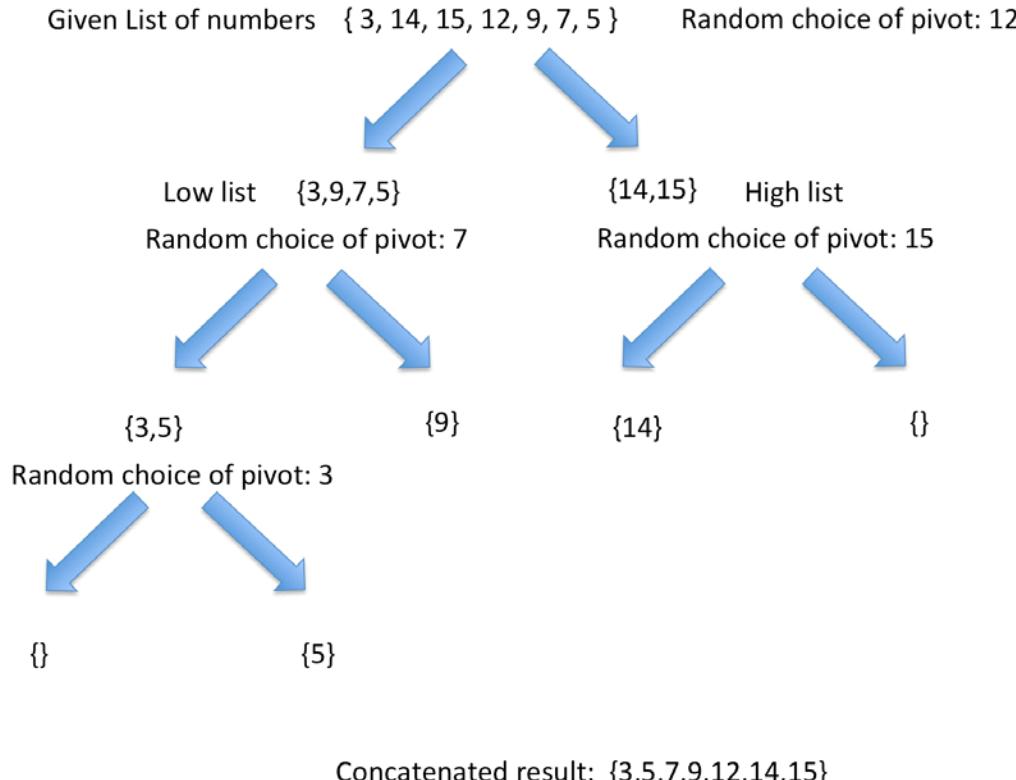


# Divide and Conquer

- Algorithms denoted as “divide and conquer” break a problem into smaller sub-problems that share similar enough algorithmic properties to the original problem that they can in turn also be subdivided
- Using recursion, the larger problem is broken down into small enough pieces that it can be easily solved with minimal computation.
- Because the original problem has been broken down into several smaller computations that are independent of one another, there is a natural concurrency for exploiting parallel computation resources
- Frequently, divide and conquer type algorithms are also naturally parallel algorithms because of this concurrency and, like fork-join type algorithms, can perform very well on shared memory architectures
- On distributed memory architectures, however, network latency and load imbalance can complicate the direct application of divide and conquer type algorithms.

# Divide and Conquer Example: Quicksort

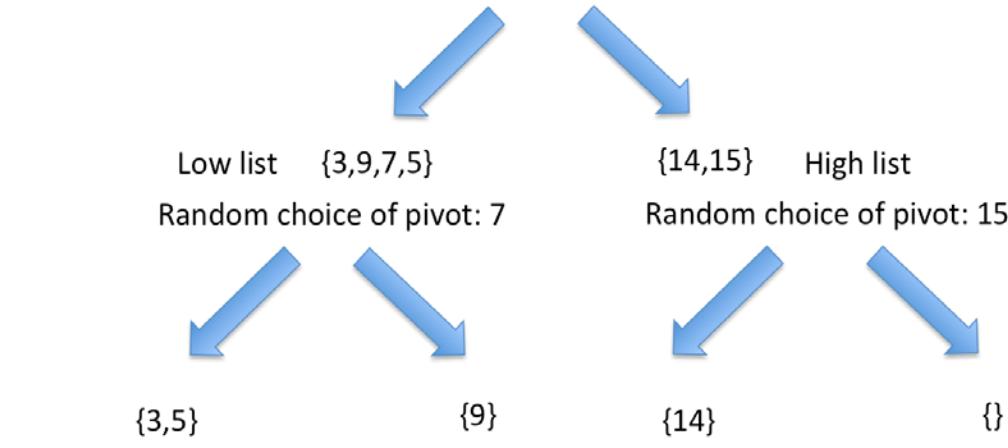
- One well studied example of a divide and conquer algorithm with natural concurrency is quicksort
- As a sorting algorithm, it aims to sort a list of numbers in order of increasing value.



*To start, a pivot point is selected: one element of the array of numbers is selected as the pivot*

# Divide and Conquer Example: Quicksort

Given List of numbers { 3, 14, 15, 12, 9, 7, 5 }      Random choice of pivot: 12

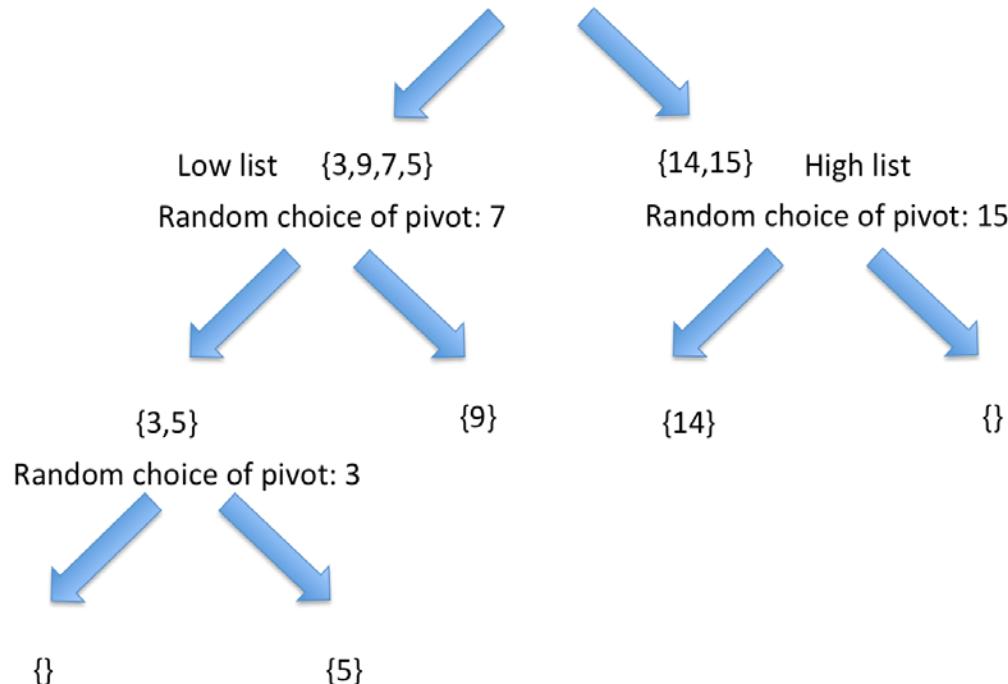


*Using this pivot,  
the rest of the list  
is divided into a  
list containing  
numbers smaller  
than the pivot and  
a list containing  
numbers larger  
than the pivot*

Concatenated result: {3,5,7,9,12,14,15}

# Divide and Conquer Example: Quicksort

Given List of numbers { 3, 14, 15, 12, 9, 7, 5 } Random choice of pivot: 12

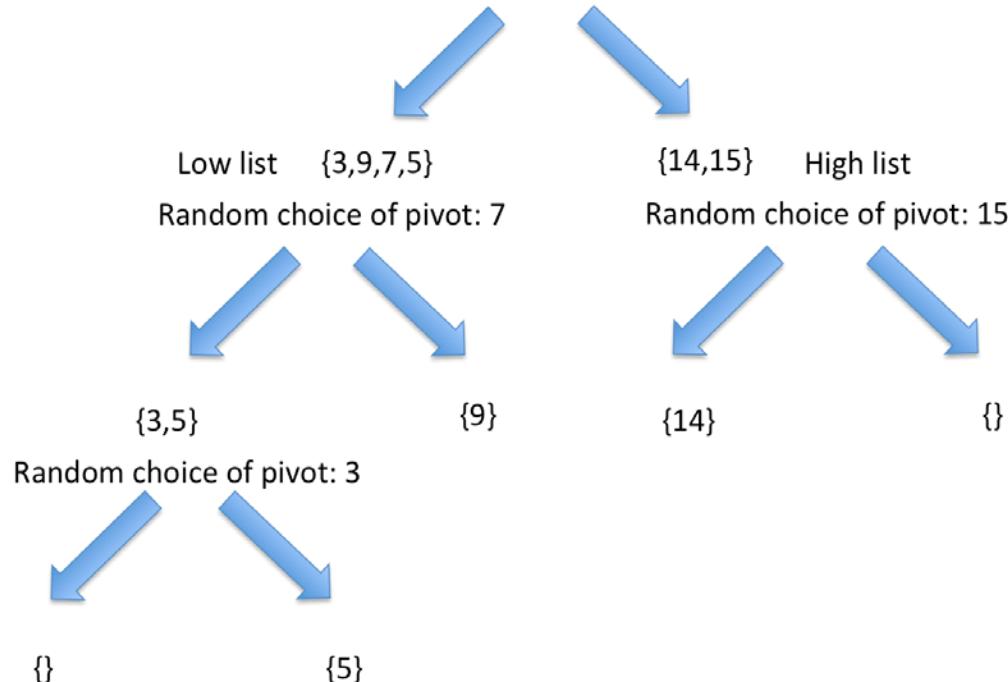


*This process is then repeated recursively for each of the two lists*

Concatenated result: {3,5,7,9,12,14,15}

# Divide and Conquer Example: Quicksort

Given List of numbers { 3, 14, 15, 12, 9, 7, 5 }      Random choice of pivot: 12



Concatenated result: {3,5,7,9,12,14,15}

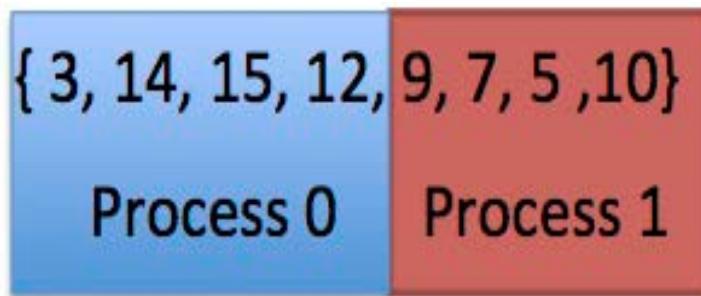
*Upon completion of recursion the resulting sorted child sub-problems are concatenated for the final result.*

# Divide and Conquer Example: Quicksort

- The efficiency of the algorithm is significantly impacted by which element is chosen as the pivot point.
- If the array has  $N$  data items, the worse case performance will be proportional to  $N^2$
- for most cases the performance is much faster, proportional to  $N \log N$
- Because the two branched lists in quicksort can be sorted independently, there is a natural concurrency of computation that can be used for parallelization

# Distributed Quicksort

- On a distributed memory architecture, exploiting this concurrency incurs a significant communication cost as sorted lists are passed from one process to another during recursion
- A modification to the approach based on sampling can be made to improve this:
  - An array of numbers to be sorted is distributed equally among P processes. Thus if the array size is N then each process will have  $N/P$  local elements.



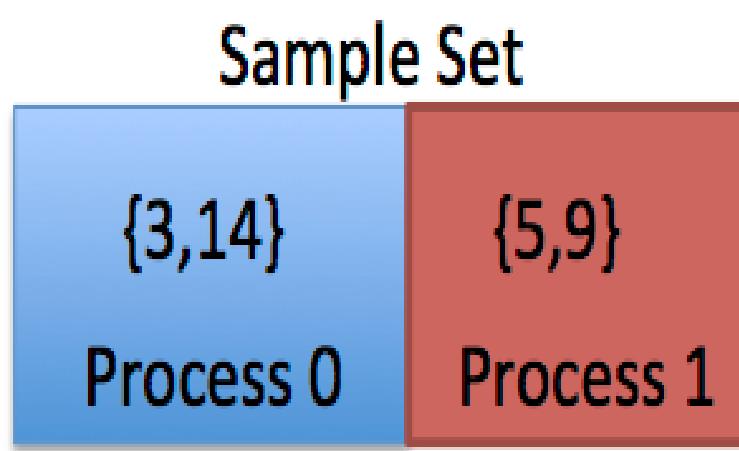
# Distributed Quicksort

- Each process runs sequential quicksort on its local data



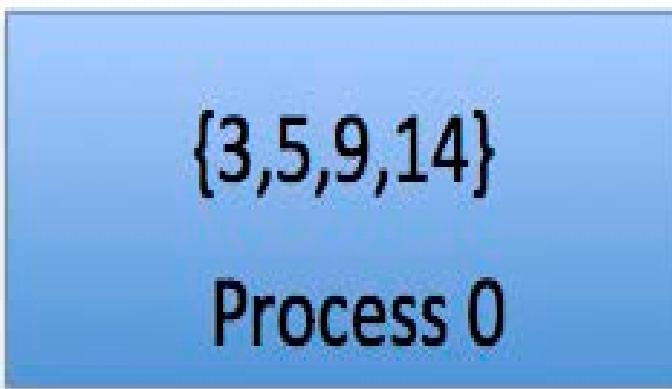
# Distributed Quicksort

- The resulting sorted arrays are sampled at intervals determined by the global array size, N, and the number of processes, P. Samples are taken at every  $N/P^2$  location starting at 0, i.e.: array element indices 0,  $N/P^2$ ,  $2N/P^2$ , ...,  $(P-1)N/P^2$  form the sample array from each local data.



# Distributed Quicksort

- The resulting samples are gathered to a root process and sorted sequentially with quicksort.

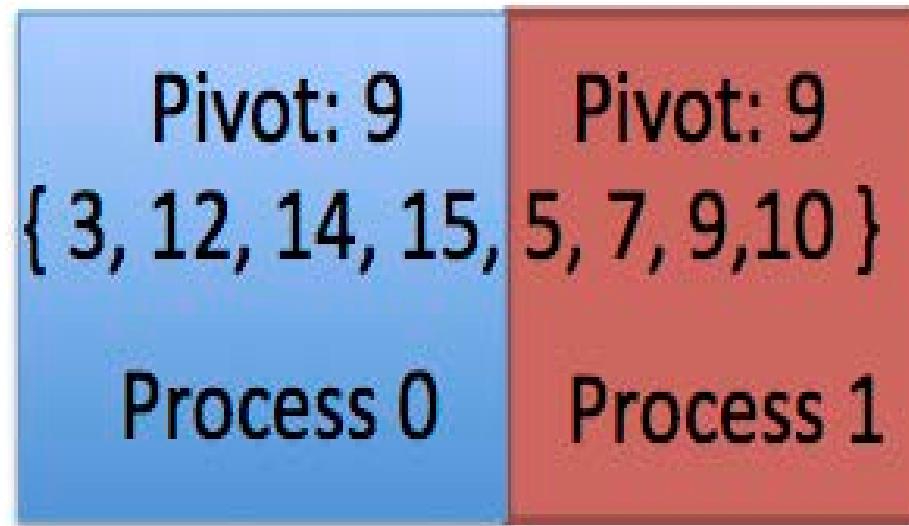


{3,5,9,14}

Process 0

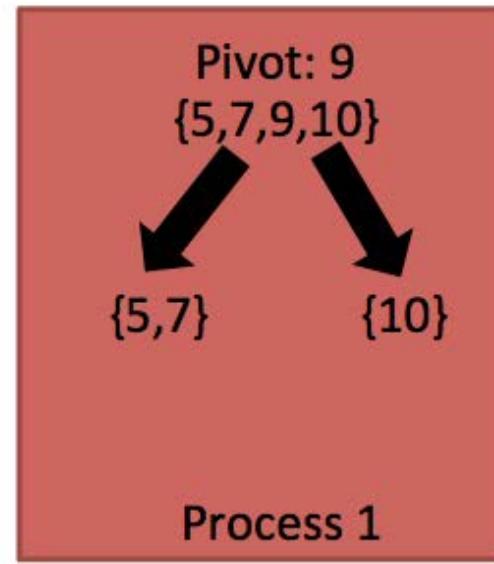
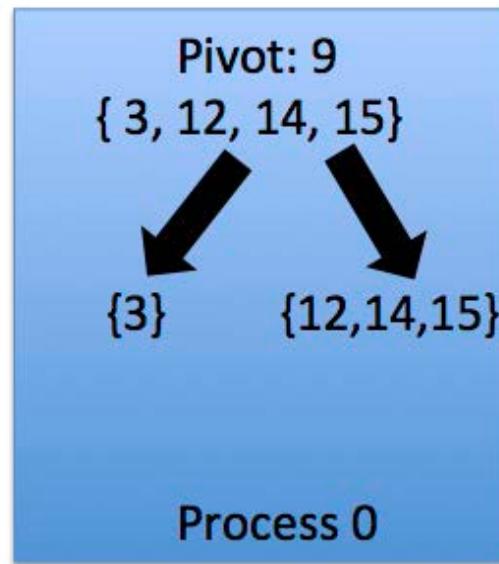
# Distributed Quicksort

- Regularly sampled  $P-1$  pivot values computed from the sample set are broadcast to the other processes. Thus  $N/P^2, 2N/P^2, \dots$ , indices form the sample  $P-1$  pivot points.



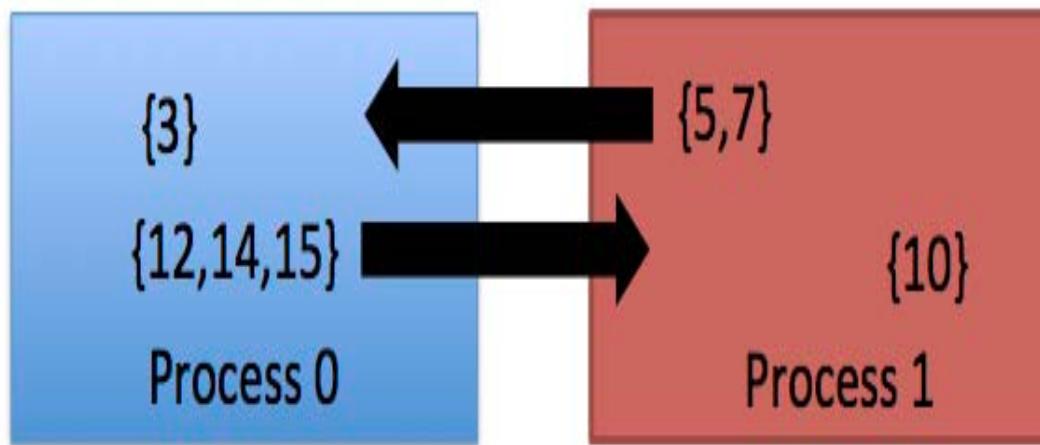
# Distributed Quicksort

- Each process divides its sorted segment of the array into P segments using the broadcasted P-1 pivot values.



# Distributed Quicksort

- Each process performs an all-to-all operation on the P segments. Thus the ith process keeps the ith segment and sends the jth segment to the jth process.



# Distributed Quicksort

- The arriving segments are merged into a single list.



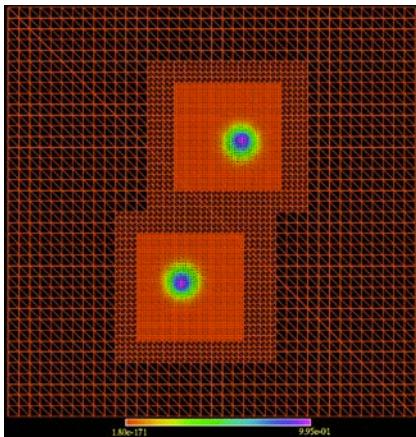
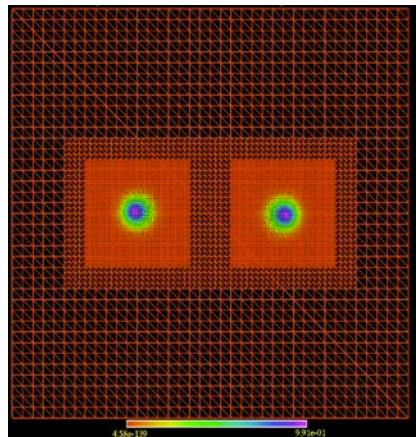
Final result:  $\{3,5,7,9,10,12,14,15\}$

# Manager Worker

- Manager worker incorporate two different workflows in their execution: one intended for execution by just one process called the manager process and another intended for execution by several other processes called worker processes.
- This approach has also historically been called “Master-Slave”
- Applications that are dynamic in nature frequently use this type of parallel design algorithm so that the manager process can coordinate and issue task actions to worker processes in response to changes in a simulation outcome.

# Manager Worker Example

```
1 if ( my_rank == master ) {  
2     send_action(INITIALIZE);  
3  
4     for (int i=0;i<num_timesteps;i++) {  
5         send_action(REFINE);  
6         send_action(INTEGRATE);  
7         send_action(OUTPUT);  
8     }  
9 } else {  
10    listen_for_actions();  
11 }
```



Manager-worker example code from an adaptive mesh refinement code. The manager process (called “master” here) directs the refinement characteristics and sends actions to the worker processes.

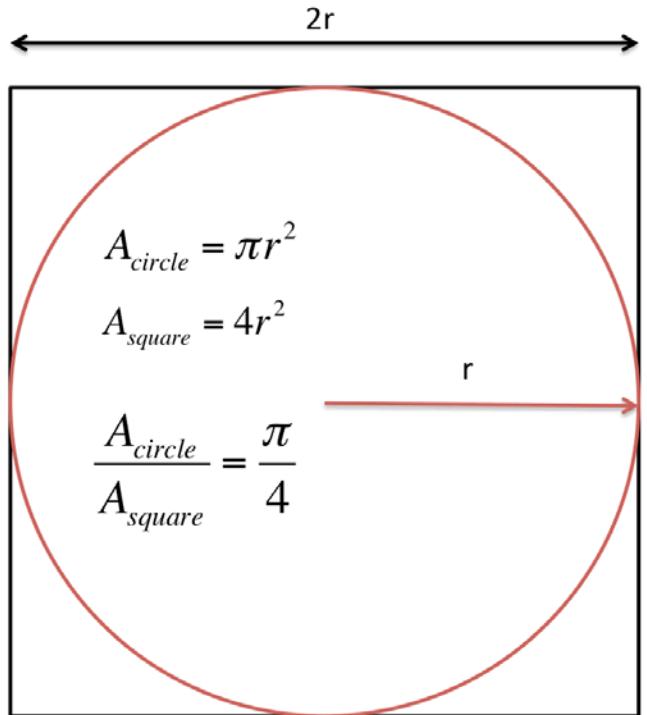
# Embarrassingly Parallel

- The term “embarrassingly parallel” is a common phrase in scientific computing that is both widely used and poorly defined
- It suggests lots of parallelism with essentially no inter-task communication or coordination as well as a highly partitionable workload with minimal overhead.
- In general, embarrassingly parallel algorithms are a subclass of manager-worker algorithms.
- They are called embarrassingly parallel because the available concurrency is trivially extracted from the workflow.
- While these algorithms may require some minimal coordination and inter-task communication, they are still generally referred to as embarrassingly parallel.

# Embarrassingly Parallel Example

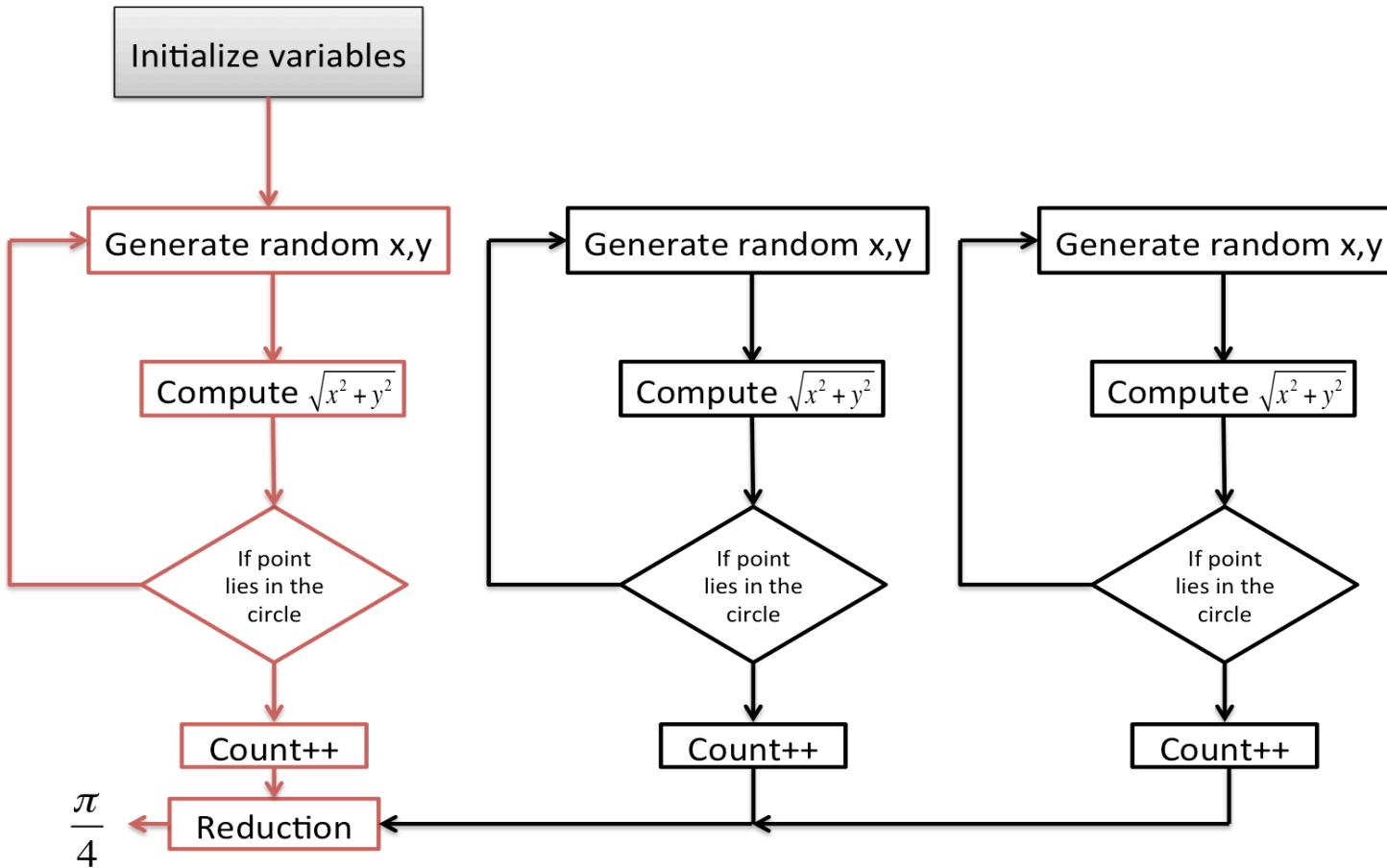
- Monte Carlo simulations generally fall into the category of embarrassingly parallel.
- Monte Carlo methods are statistical approaches for studying systems with:
  - a large number of coupled degrees of freedom,
  - modeling phenomena with significant uncertainty in the inputs,
  - and solving partial differential equations with more than 4 dimensions.
- Computing the value of pi is a simple example

# Embarrassingly Parallel Example: Pi



- Define a square domain and inscribe a circle inside that domain
- Randomly generate the coordinates of points lying inside the square domain; count the points that also lie in the circle.
- $\pi/4$  is the ratio of the number of points that lie in the circle to the total number of random points generated.

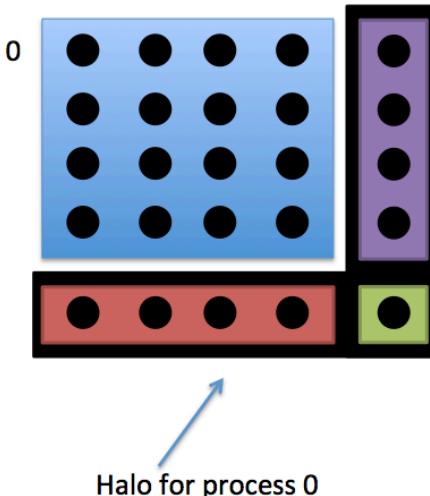
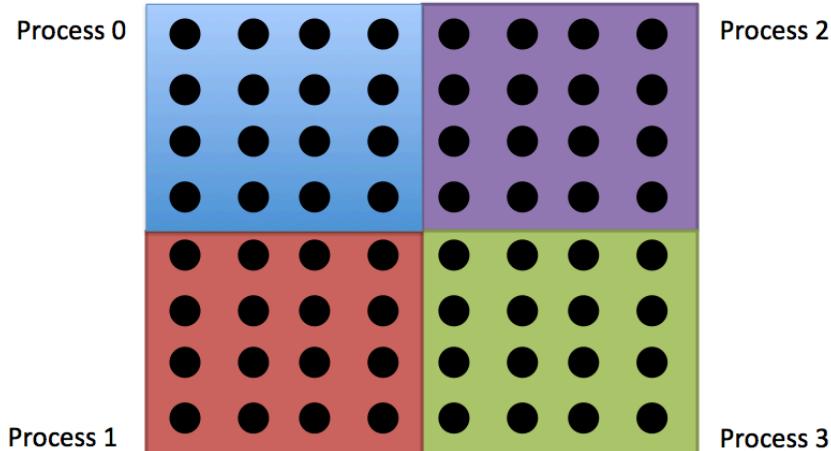
# Embarrassingly Parallel Example: Pi



# Halo-exchange

- Many parallel algorithms fall into a parallel problem class where every parallel task is executing the same algorithm on different data without any manager algorithm present.
- This is sometimes referred to as the data parallel model.
- Data parallelism is frequently used in applications that are static in nature because a computational task can be mapped to particular subset of data throughout the life of the simulation.
- Some information in each data subset mapped to the parallel task has to be exchanged and synchronized in order for the application algorithm to function properly.
- This exchange of inter-task information is called *halo-exchange*.

# Halo Exchange



- As the name implies, a halo is a region exterior to the data subset mapped to a parallel task.
- It acts as an artificial boundary to that data subset and contains information that originates from the data subsets of neighboring parallel tasks.

# Halo Exchange Example – The Advection Equation

- The study of wavelike phenomena is ubiquitous on supercomputing systems and is frequently modeled using a *partial differential equation*, or an expression involving derivatives taken against different independent variables.
- One of the simplest ways to solve these wavelike partial differential equations on a supercomputer is through the use of finite differencing and halo exchange.
- Finite differencing involves replacing the derivative expressions in the partial differential equation with approximations originating from estimating the slope between neighboring points on a uniform grid.
- As an example of this parallel algorithm, consider the advection equation :

$$\frac{\partial f}{\partial t} = -v \frac{\partial f}{\partial x}$$

# The Advection Equation

$$\frac{\partial f}{\partial t} = -v \frac{\partial f}{\partial x}$$

- This advection equation transports a scalar field  $f(x,t)$  towards increasing  $x$  value with speed  $v$ .
- The analytic solution to this partial differential equation is

$$f(x,t) = F(x - vt)$$

Where  $F(x)$  is an arbitrary function describing the initial condition of the system.

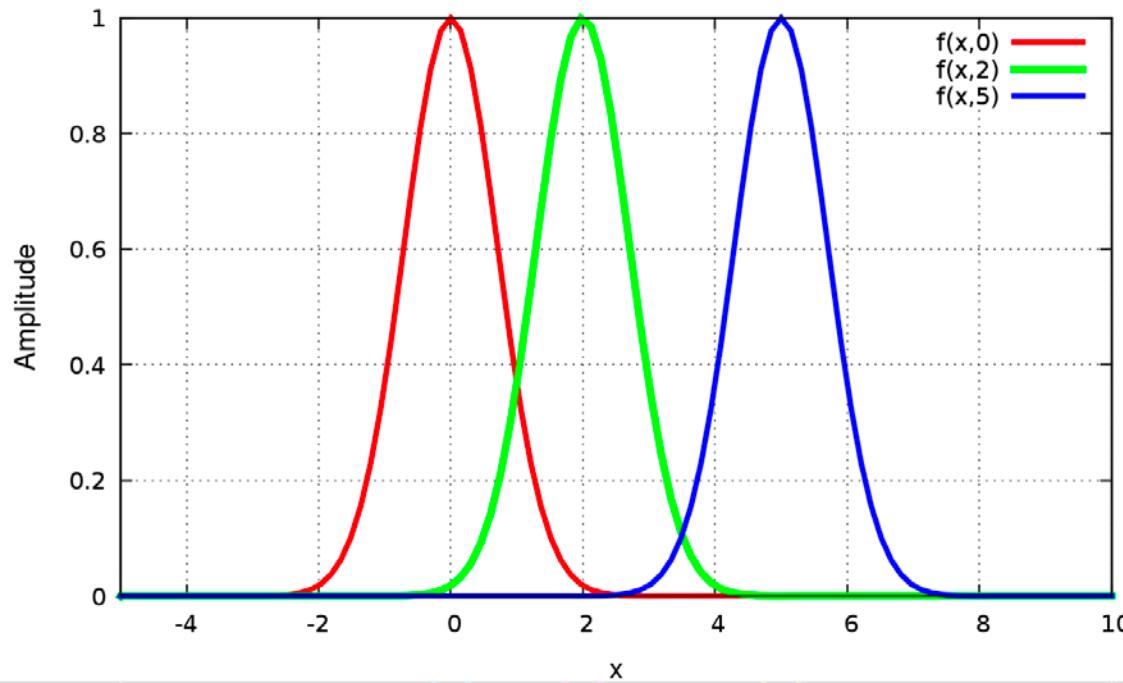
# The Advection Equation

- Suppose the initial condition was:

$$F(x) = e^{-x^2}$$

- The analytic solution to the advection equation would be:

$$f(x, t) = e^{-(x-vt)^2}$$

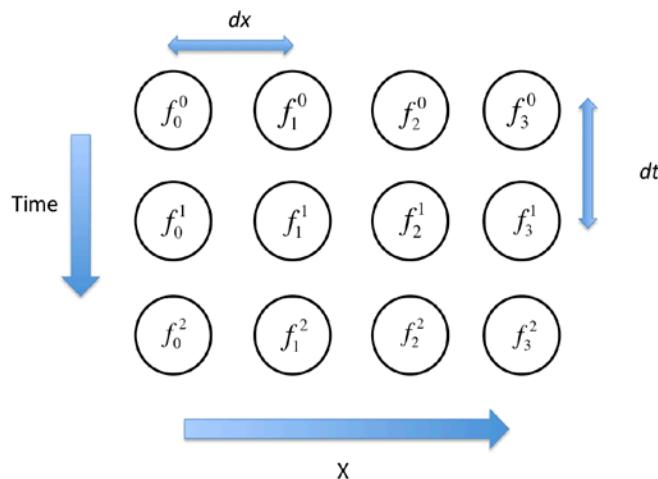


# Halo Exchange – the advection equation

- The left and right hand partial derivatives in the advection equation are replaced with finite difference approximations to those derivatives

$$\frac{f_i^{n+1} - f_i^n}{dt} = -v \frac{f_{i+1}^n - f_i^n}{dx}$$

- The field  $f(x,t)$  has been discretized to a uniform mesh where the  $x$  points are separated by distance  $dx$  and the time points are separated by time  $dt$  with the subscript to  $f$  indicating the spatial location in that mesh and the superscript to  $f$  indicating the temporal location in that mesh.

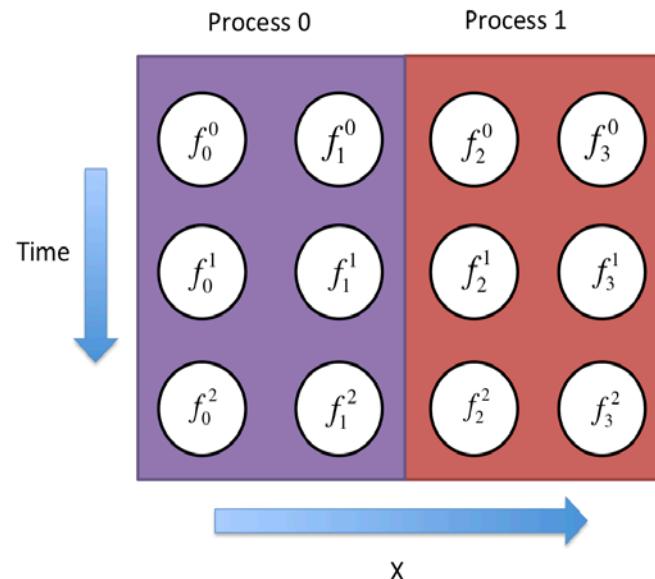


# Halo Exchange – the advection equation

- Algebraic manipulation of the finite difference discretization enables all future times values to be found iteratively:

$$f_i^{n+1} = f_i^n - v \frac{dt}{dx} \left( \frac{f_{i+1}^n - f_i^n}{dx} \right)$$

- In order to compute the right hand side of this expression in parallel, the discretized mesh is partitioned across several processes:

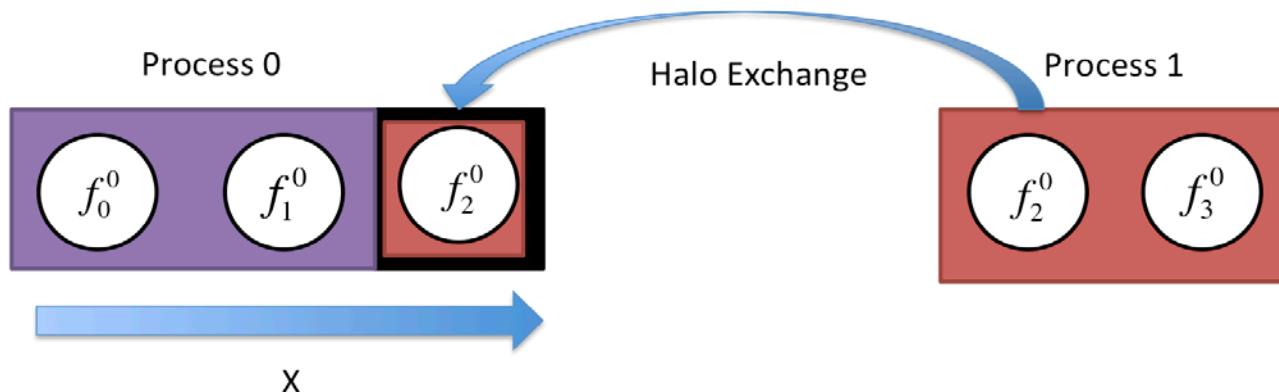


# Halo Exchange – the advection equation

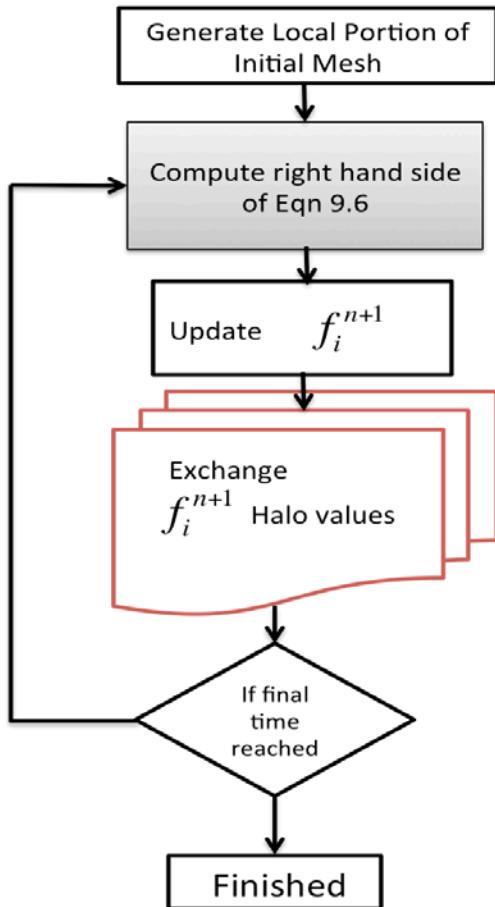
- To evaluate the right hand side of

$$f_i^{n+1} = f_i^n - v \frac{dt}{dx} \left( \frac{f_{i+1}^n - f_i^n}{dx} \right) \quad (9.6)$$

for process 0, some information is needed from process 1. This information is provided through halo exchange:



# Halo Exchange – the advection equation



# Halo Exchange – Sparse Matrix Vector Multiplication

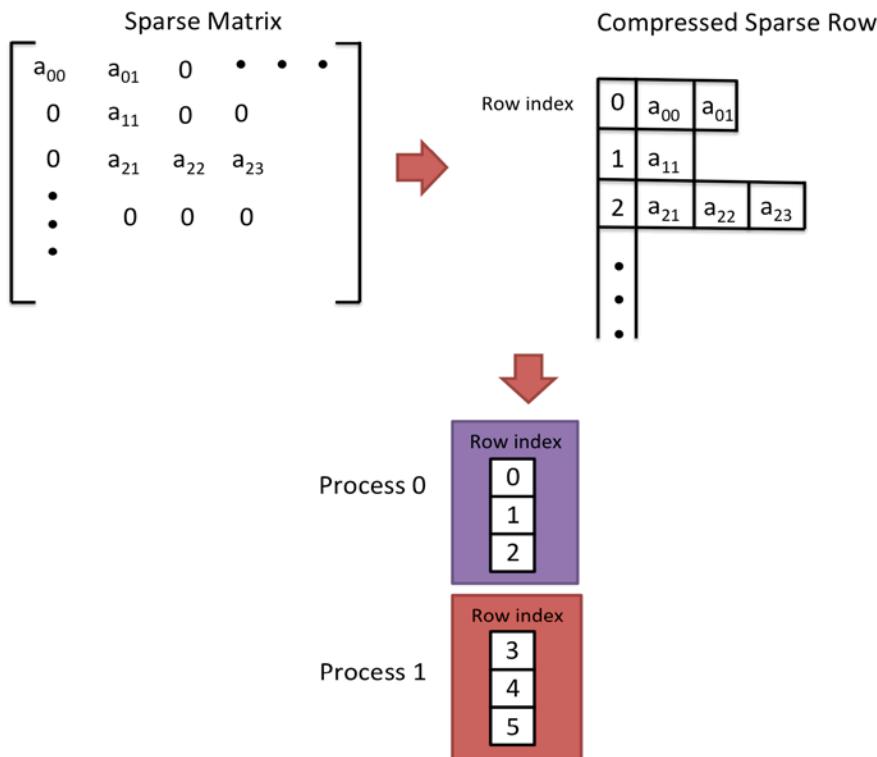
- Parallel algorithms designed around halo exchange frequently show up not just in mesh based solvers but also in sparse linear algebra operations such as sparse matrix vector multiplication used in the HPCG benchmark.
- For a matrix of size  $N \times N$  and vector of size  $N$ , matrix-vector multiplication is given by

$$x_i = \sum_{j=0}^{N-1} A_{ij} b_j$$

where  $A_{ij}$  is the  $(i,j)$ -th element of the matrix and  $b_j$  is the  $j$ -th element of the vector.

# Halo Exchange – Sparse Matrix Vector Multiplication

- For a sparse matrix most of the  $A_{ij}$  values are zero suggesting that memory would not need to be allocated for every  $(i,j)$ -th element.
- To avoid storing or manipulating zero entries, the *compressed sparse rows* format is frequently used:



For each row, all nonzeros of that row are placed in contiguous memory.

In order to achieve data parallelism, the matrix and vector are partitioned into groups of rows assigned to each process.

# Halo Exchange – Sparse Matrix Vector Multiplication

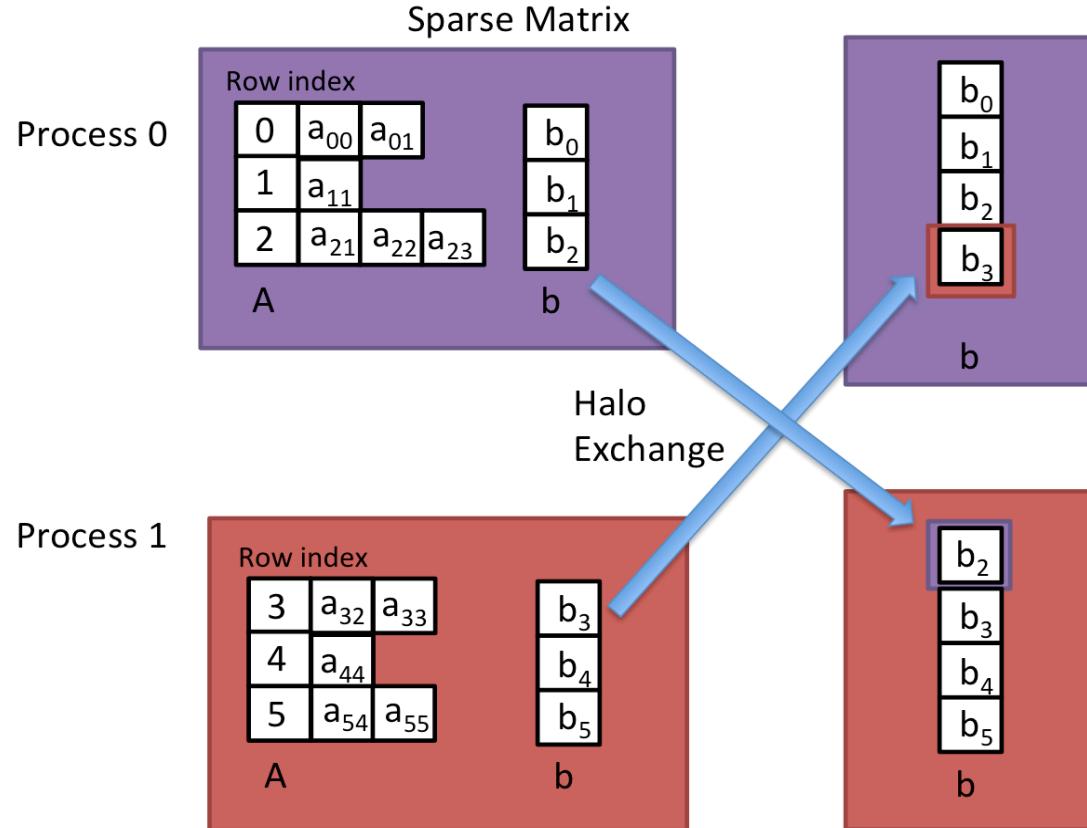
- The compressed sparse format alters the matrix vector multiplication formula to reflect that zeros are no longer stored nor manipulated:

$$x_i = \sum_{j=0}^{n_i} A_{ij} b_j \quad (9.8)$$

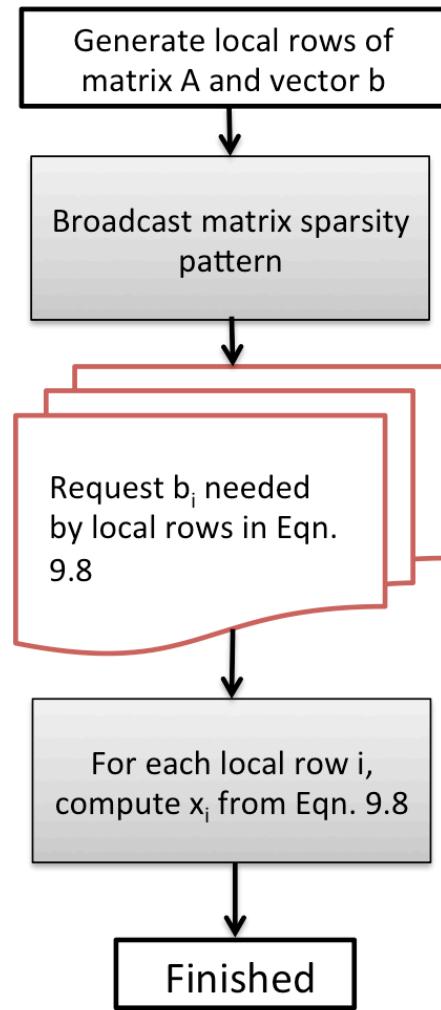
where  $n_i$  indicates the number of nonzeros for the  $i$ -th row.

- Because the rows for both matrix A and vector b are partitioned across several processes, some information will need to be exchanged to evaluate this expression over different sets of data.

# Halo Exchange – Sparse Matrix Vector Multiplication



# Halo Exchange – Sparse Matrix Vector Multiplication



# Permutation: Cannon's Algorithm

- Among algorithms which rely upon a data parallelism approach where the same algorithm is applied to different data in order to extract concurrency, a certain subclass of problem relies upon iterative permutation routing operations to perform all-to-all operations iteratively.
- This type of parallel algorithm is very frequently used in applications requiring a linear algebra transpose operation or some type of matrix-matrix multiplication.
- Here we explore one such example: Cannon's algorithm for dense matrix-matrix.

# Cannon's Algorithm

- Matrix-matrix multiplication for two  $N \times N$  matrices A and B is given by:

$$C_{ij} = \sum_{k=0}^{k=N-1} A_{ik} B_{kj}$$

where the subscripts indicate the row and column index of the matrix entry.

- In order to create a parallel algorithm for this, a good place to start is a block algorithm that distributes sub-blocks of matrices A, B, and C among processes where each sub-block is of size  $N/P \times N/P$  where P is the number of processes

# Cannon's Algorithm

*distribute sub-blocks of A, B, and C among processes where each sub-block is of size  $N/P \times N/P$  where P is the number of processes*

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

=

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

# Cannon's Algorithm

- In order to compute the sub-block  $C_{11}$  of the matrix-matrix product of  $A^*B$  would require computing several serial matrix-matrix products each of size  $N/\sqrt{P} \times N/\sqrt{P}$

The diagram illustrates the computation of the sub-block  $C_{11}$  of the matrix-matrix product  $A^*B$ . It shows two 4x4 matrices, A and B, and their product C. Matrix A is partitioned into four 2x2 sub-blocks:  $A_{00}$ ,  $A_{01}$ ,  $A_{02}$ ,  $A_{03}$  in the top row; and  $A_{10}$ ,  $A_{11}$ ,  $A_{12}$ ,  $A_{13}$  in the bottom row. Matrix B is partitioned into four 2x2 sub-blocks:  $B_{00}$ ,  $B_{01}$ ,  $B_{02}$ ,  $B_{03}$  in the top row; and  $B_{10}$ ,  $B_{11}$ ,  $B_{12}$ ,  $B_{13}$  in the bottom row. Matrix C is partitioned into four 2x2 sub-blocks:  $C_{00}$ ,  $C_{01}$ ,  $C_{02}$ ,  $C_{03}$  in the top row; and  $C_{10}$ ,  $C_{11}$ ,  $C_{12}$ ,  $C_{13}$  in the bottom row. The sub-block  $C_{11}$  is highlighted in orange. The equation  $C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$  is shown below the matrices.

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

=

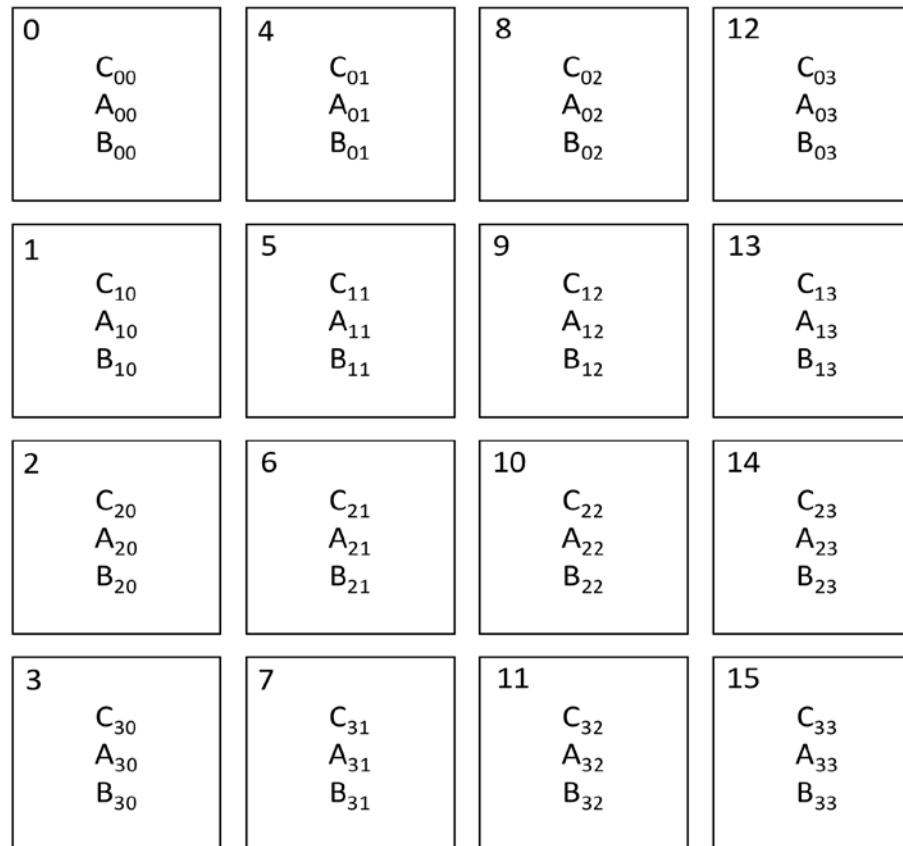
$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

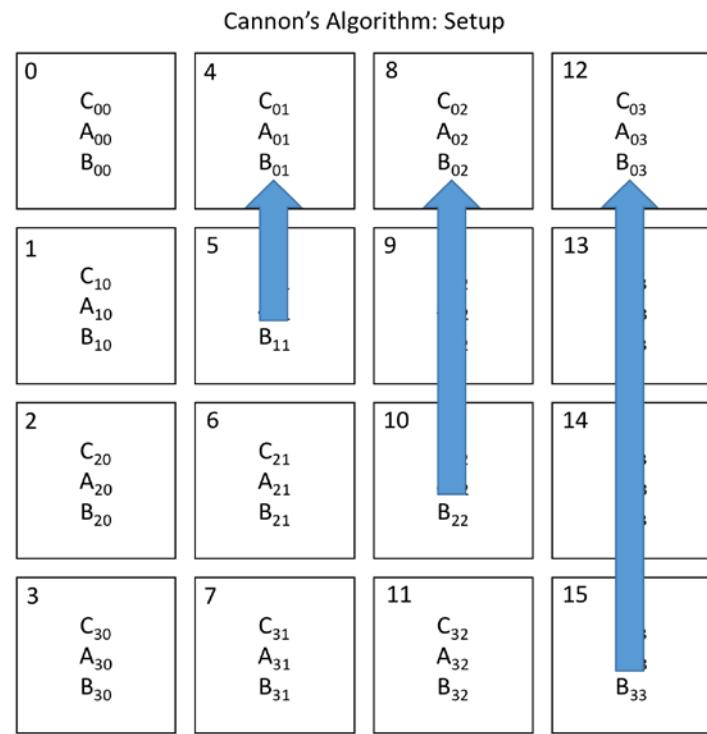
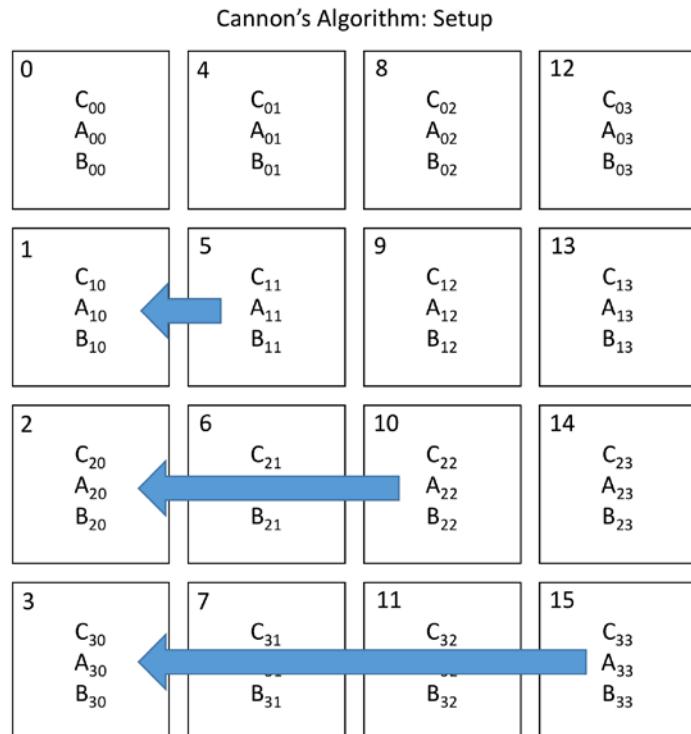
# Cannon's Algorithm

- Initially, sub-blocks are mapped to each process as illustrated:



# Cannon's Algorithm

- To setup Cannon's algorithm, the A sub-blocks are shifted to the left while the B sub-blocks are shifted up:



# Cannon's Algorithm

- The layout of the matrix sub-blocks after performing the setup permutations

0 $C_{00}$ $A_{00}$ $B_{00}$	4 $C_{01}$ $A_{01}$ $B_{11}$	8 $C_{02}$ $A_{02}$ $B_{22}$	12 $C_{03}$ $A_{03}$ $B_{33}$
1 $C_{10}$ $A_{11}$ $B_{10}$	5 $C_{11}$ $A_{12}$ $B_{21}$	9 $C_{12}$ $A_{13}$ $B_{32}$	13 $C_{13}$ $A_{10}$ $B_{03}$
2 $C_{20}$ $A_{22}$ $B_{20}$	6 $C_{21}$ $A_{23}$ $B_{31}$	10 $C_{22}$ $A_{20}$ $B_{02}$	14 $C_{23}$ $A_{21}$ $B_{13}$
3 $C_{30}$ $A_{33}$ $B_{30}$	7 $C_{31}$ $A_{30}$ $B_{01}$	11 $C_{32}$ $A_{31}$ $B_{12}$	15 $C_{33}$ $A_{32}$ $B_{23}$

# Cannon's Algorithm

- Cannon's algorithm consists of moving matrix sub-blocks so that for each iteration  $k$  from 0 to 3 matrix sub-blocks  $A_{i, (i+j+k)}$  and  $B_{(i+j+k),j}$  are located on the same process as  $C_{ij}$ .
- For each iteration, the partial sum

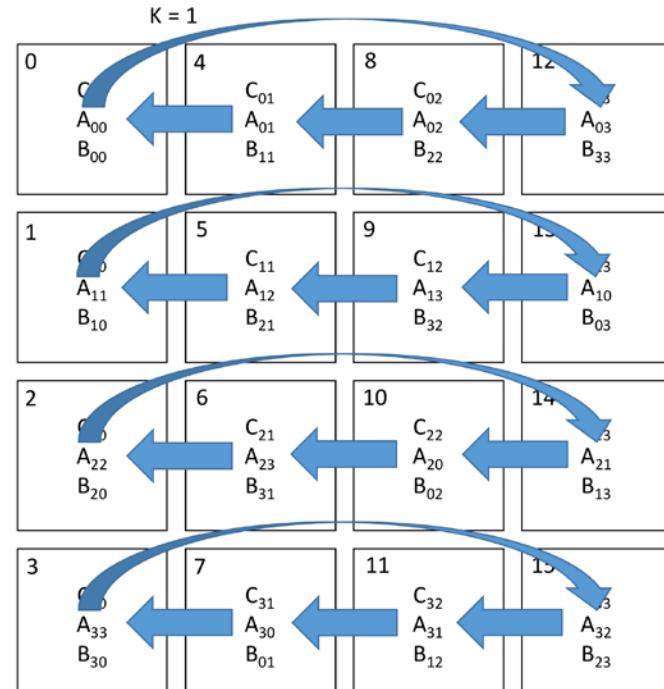
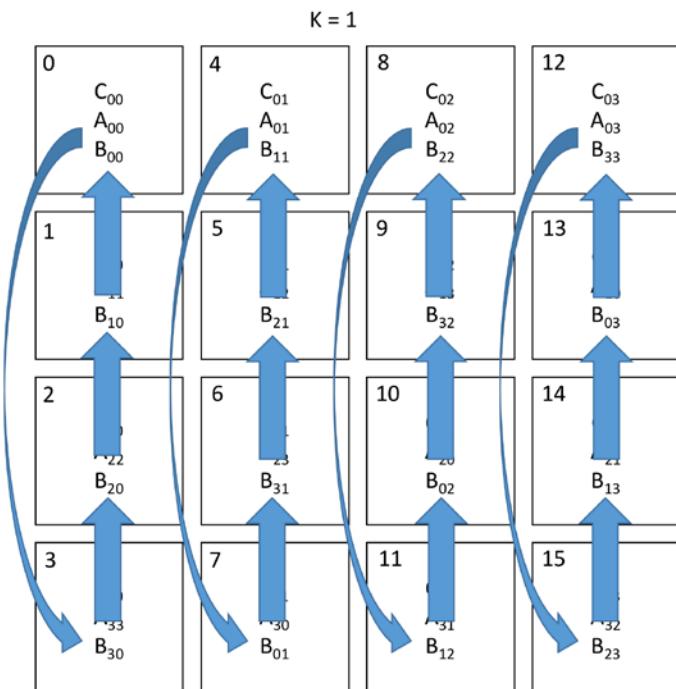
$$C_{ij} += A_{i,(i+j+k)} B_{(i+j+k),j} \quad (9.9)$$

is accumulated to  $C_{ij}$  where each sub-block matrix-matrix multiplication uses the serial matrix-matrix multiplication algorithm previously shown.

- The sums in Eqn. 9.9,  $i+j+k$  are modulus  $P$  (4 in this example). Thus if  $(i+j+k) = 6$ , the index in the matrix would become 2.

# Cannon's Algorithm

- For  $k=0$ , Cannon's algorithm has already been setup.
- For every subsequent iteration of  $k$ , the  $A$  matrices have to be shifted once left and the  $B$  matrices have to be shifted up once to satisfy the condition of Eqn. 9.9 and compute the partial sum.

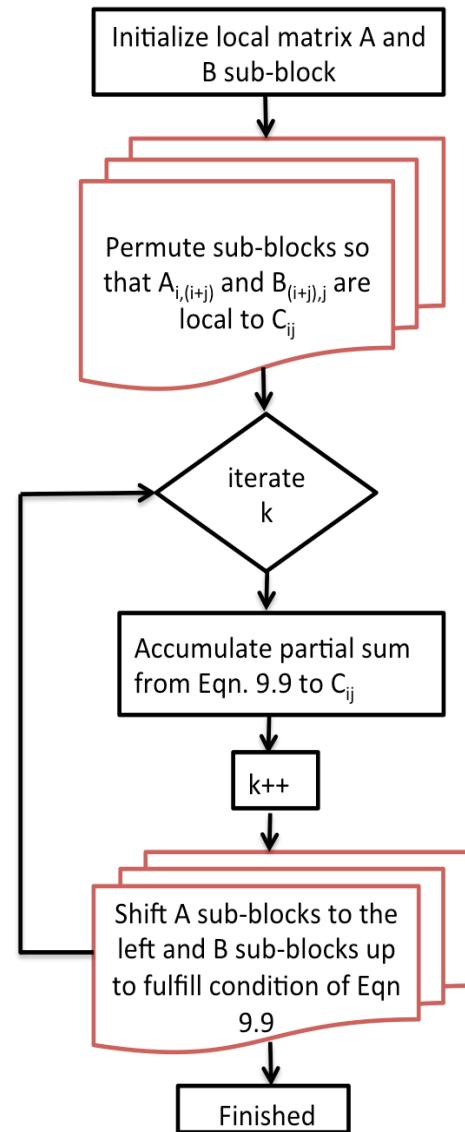


# Cannon's Algorithm

- After  $P$  iterations of  $k$ , the matrix-matrix product has been computed. The resulting matrices for each of the  $k$  iterations for  $P=4$  are shown:

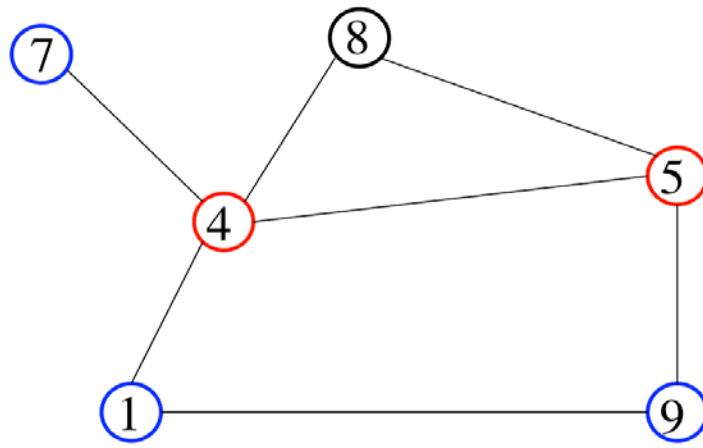
K = 0				K = 1				K = 2				K = 3			
0 $C_{00}$ $A_{00}$ $B_{00}$	4 $C_{01}$ $A_{01}$ $B_{11}$	8 $C_{02}$ $A_{02}$ $B_{22}$	12 $C_{03}$ $A_{03}$ $B_{33}$	0 $C_{00}$ $A_{01}$ $B_{10}$	4 $C_{01}$ $A_{02}$ $B_{21}$	8 $C_{02}$ $A_{03}$ $B_{32}$	12 $C_{03}$ $A_{00}$ $B_{03}$	0 $C_{00}$ $A_{02}$ $B_{20}$	4 $C_{01}$ $A_{12}$ $B_{31}$	8 $C_{12}$ $A_{13}$ $B_{02}$	12 $C_{13}$ $A_{10}$ $B_{13}$	0 $C_{00}$ $A_{03}$ $B_{30}$	4 $C_{01}$ $A_{00}$ $B_{01}$	8 $C_{02}$ $A_{01}$ $B_{12}$	12 $C_{03}$ $A_{02}$ $B_{23}$
1 $C_{10}$ $A_{11}$ $B_{10}$	5 $C_{11}$ $A_{12}$ $B_{21}$	9 $C_{12}$ $A_{13}$ $B_{32}$	13 $C_{13}$ $A_{10}$ $B_{03}$	1 $C_{10}$ $A_{12}$ $B_{20}$	5 $C_{11}$ $A_{13}$ $B_{31}$	9 $C_{12}$ $A_{10}$ $B_{02}$	13 $C_{13}$ $A_{11}$ $B_{13}$	2 $C_{20}$ $A_{22}$ $B_{20}$	6 $C_{21}$ $A_{23}$ $B_{31}$	10 $C_{22}$ $A_{20}$ $B_{02}$	14 $C_{23}$ $A_{21}$ $B_{13}$	2 $C_{20}$ $A_{23}$ $B_{30}$	6 $C_{21}$ $A_{20}$ $B_{01}$	10 $C_{22}$ $A_{21}$ $B_{12}$	14 $C_{23}$ $A_{22}$ $B_{23}$
2 $C_{20}$ $A_{22}$ $B_{20}$	6 $C_{21}$ $A_{23}$ $B_{31}$	10 $C_{22}$ $A_{20}$ $B_{02}$	14 $C_{23}$ $A_{21}$ $B_{13}$	3 $C_{30}$ $A_{33}$ $B_{30}$	7 $C_{31}$ $A_{30}$ $B_{01}$	11 $C_{32}$ $A_{31}$ $B_{12}$	15 $C_{33}$ $A_{32}$ $B_{23}$	3 $C_{30}$ $A_{30}$ $B_{00}$	7 $C_{31}$ $A_{31}$ $B_{11}$	11 $C_{32}$ $A_{32}$ $B_{22}$	15 $C_{33}$ $A_{33}$ $B_{33}$	3 $C_{30}$ $A_{31}$ $B_{10}$	7 $C_{31}$ $A_{32}$ $B_{21}$	11 $C_{32}$ $A_{33}$ $B_{02}$	15 $C_{33}$ $A_{31}$ $B_{13}$
K = 2				K = 3				K = 0				K = 1			
0 $C_{00}$ $A_{02}$ $B_{20}$	4 $C_{01}$ $A_{03}$ $B_{31}$	8 $C_{02}$ $A_{00}$ $B_{02}$	12 $C_{03}$ $A_{01}$ $B_{13}$	0 $C_{00}$ $A_{03}$ $B_{30}$	4 $C_{01}$ $A_{00}$ $B_{01}$	8 $C_{02}$ $A_{01}$ $B_{12}$	12 $C_{03}$ $A_{02}$ $B_{23}$	1 $C_{10}$ $A_{13}$ $B_{30}$	5 $C_{11}$ $A_{10}$ $B_{01}$	9 $C_{12}$ $A_{11}$ $B_{12}$	13 $C_{13}$ $A_{12}$ $B_{23}$	1 $C_{10}$ $A_{10}$ $B_{00}$	5 $C_{11}$ $A_{11}$ $B_{11}$	9 $C_{12}$ $A_{12}$ $B_{22}$	13 $C_{13}$ $A_{13}$ $B_{33}$
1 $C_{10}$ $A_{13}$ $B_{30}$	5 $C_{11}$ $A_{10}$ $B_{01}$	9 $C_{12}$ $A_{11}$ $B_{12}$	13 $C_{13}$ $A_{12}$ $B_{23}$	2 $C_{20}$ $A_{20}$ $B_{00}$	6 $C_{21}$ $A_{21}$ $B_{11}$	10 $C_{22}$ $A_{22}$ $B_{22}$	14 $C_{23}$ $A_{23}$ $B_{33}$	2 $C_{20}$ $A_{21}$ $B_{10}$	6 $C_{21}$ $A_{22}$ $B_{21}$	10 $C_{22}$ $A_{23}$ $B_{32}$	14 $C_{23}$ $A_{20}$ $B_{03}$	3 $C_{30}$ $A_{31}$ $B_{10}$	7 $C_{31}$ $A_{32}$ $B_{21}$	11 $C_{32}$ $A_{33}$ $B_{02}$	15 $C_{33}$ $A_{31}$ $B_{13}$
2 $C_{20}$ $A_{20}$ $B_{00}$	6 $C_{21}$ $A_{21}$ $B_{11}$	10 $C_{22}$ $A_{22}$ $B_{22}$	14 $C_{23}$ $A_{23}$ $B_{33}$	3 $C_{30}$ $A_{31}$ $B_{10}$	7 $C_{31}$ $A_{32}$ $B_{21}$	11 $C_{32}$ $A_{33}$ $B_{32}$	15 $C_{33}$ $A_{30}$ $B_{03}$	3 $C_{30}$ $A_{32}$ $B_{20}$	7 $C_{31}$ $A_{33}$ $B_{31}$	11 $C_{32}$ $A_{30}$ $B_{02}$	15 $C_{33}$ $A_{31}$ $B_{13}$	3 $C_{30}$ $A_{33}$ $B_{20}$	7 $C_{31}$ $A_{31}$ $B_{11}$	11 $C_{32}$ $A_{32}$ $B_{01}$	15 $C_{33}$ $A_{33}$ $B_{13}$

# Cannon's Algorithm



# Task Dataflow: Breadth first search

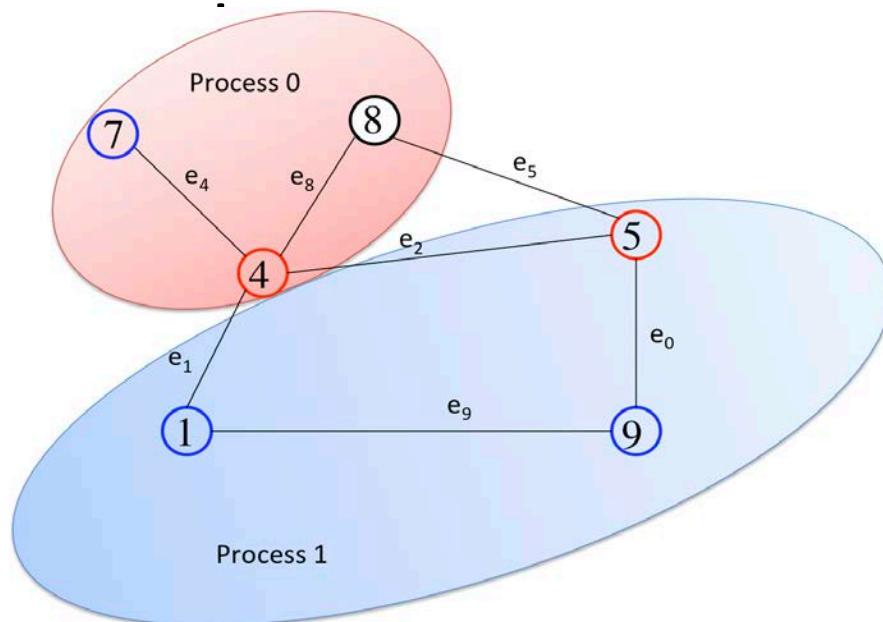
- The breadth first search algorithm is used for traversing graph data structures and is a key component of the Graph500 benchmark
- A particular root vertex is given to the algorithm to start traversing the graph data structure.
- Each adjacent vertex to the root is then traversed and so on thereby establishing the level (or distance) of every vertex from the root.



Starting at root 8: 8, 4, 5, 1, 7, 9

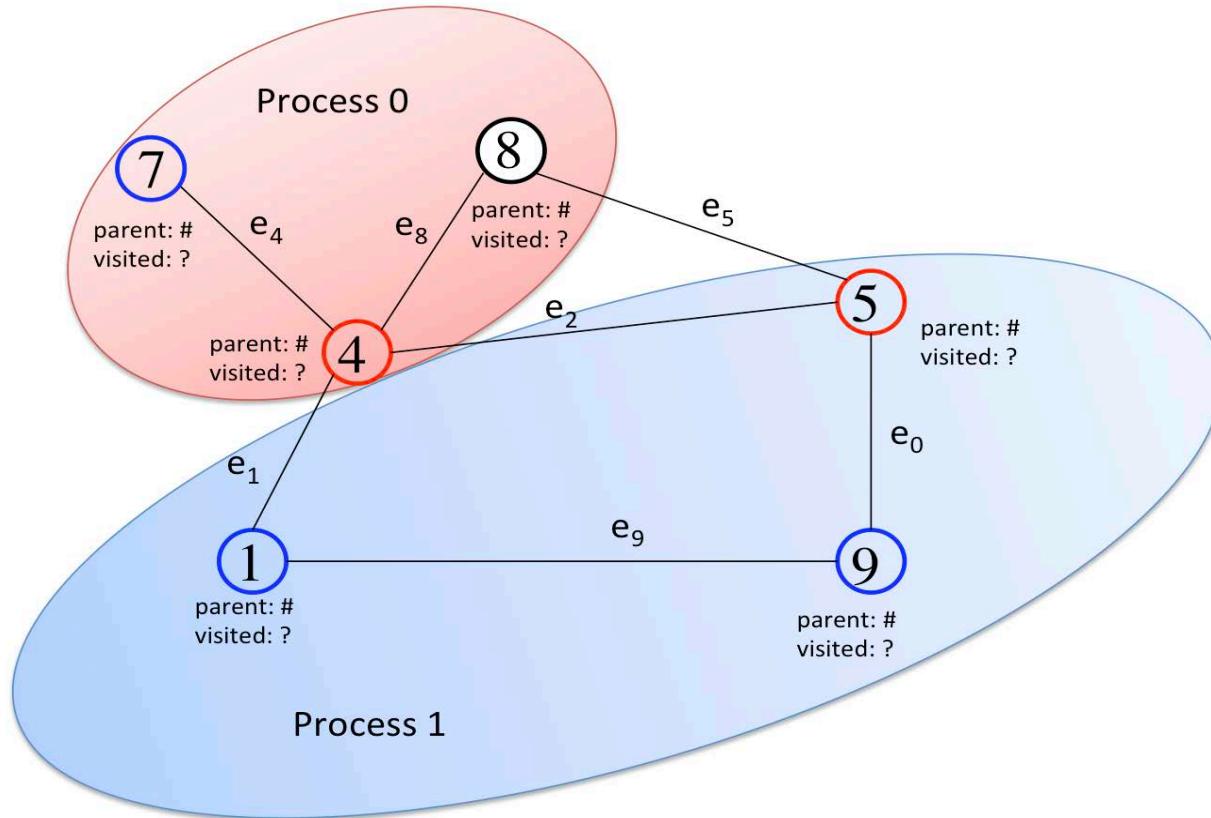
# Task Dataflow: Breadth first search

- While any parallel algorithm can be expressed as a graph of dependencies, many algorithms that explore graphs themselves are naturally expressed as task dataflow to maximize concurrency.
- We use standard parallel breadth first search algorithm as one such



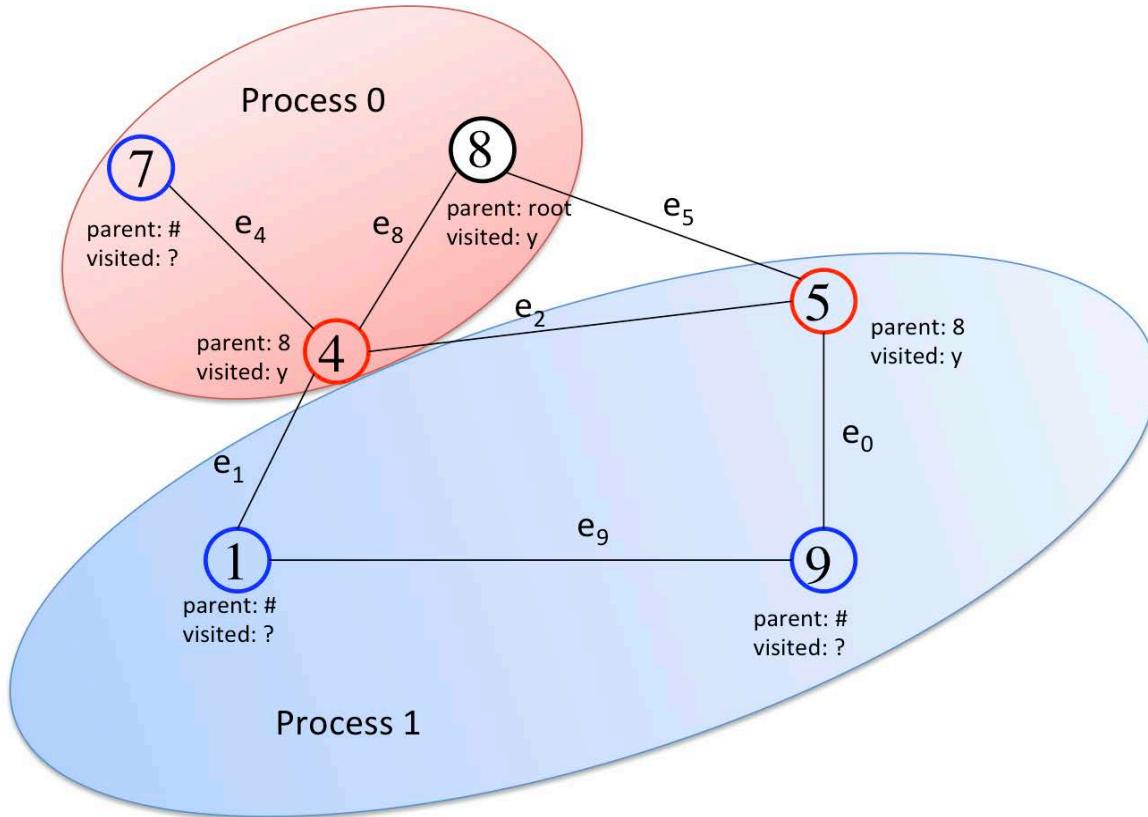
*Each vertex list is partitioned by process with its edge list*

# Breadth first search



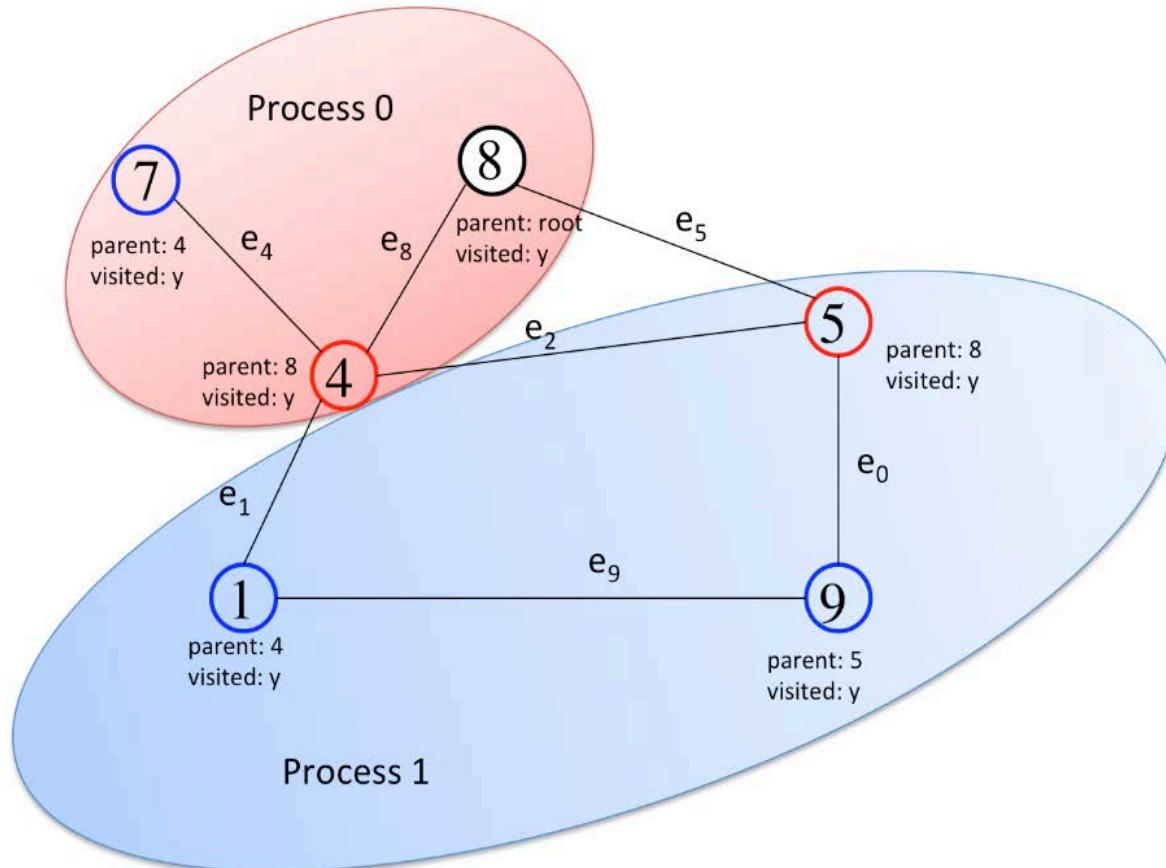
*For each vertex, associate a parent vertex and a binary flag indicating if the vertex has been visited*

# Breadth first search



*On each process,  
scan if new  
vertices are  
visited*

# Breadth first search



*For each process new vertex visited, follow the edge list and if the vertex is unvisited, set the parent and set to visited*

# Breadth first search

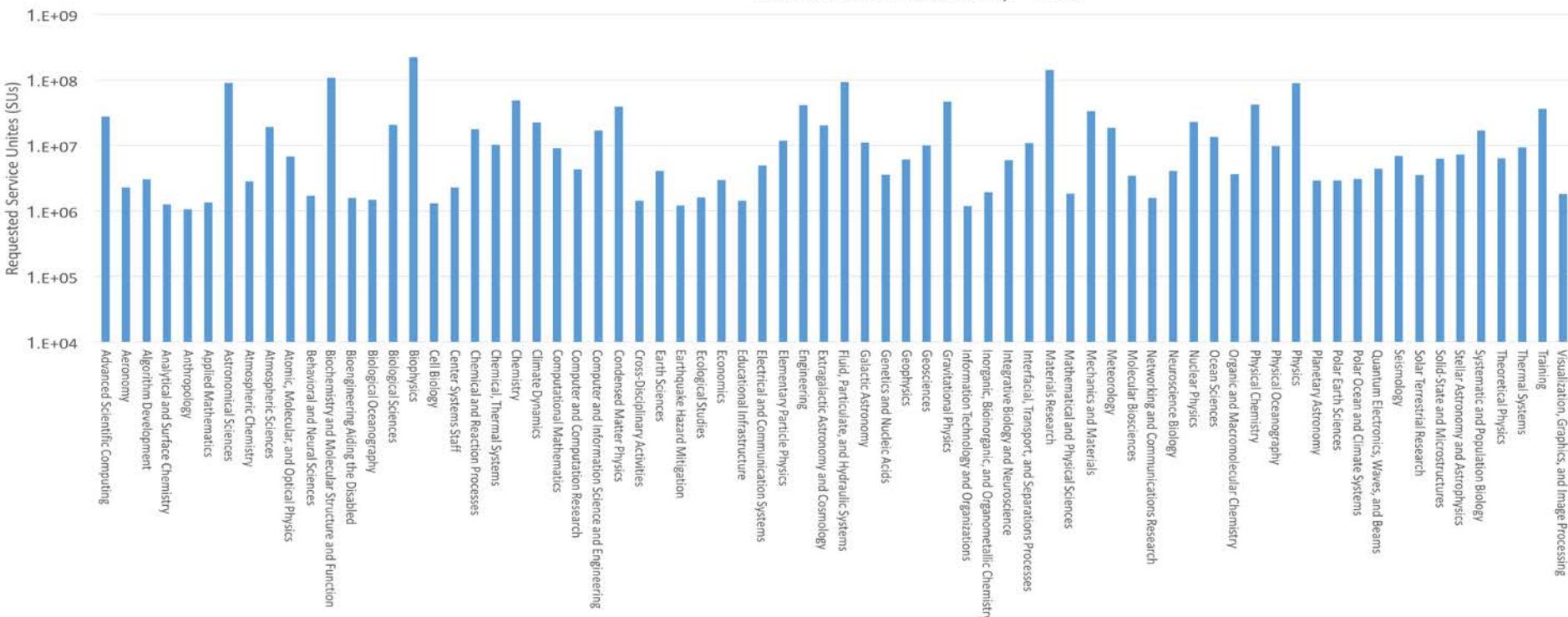
- Level-wise iteration is enforced with two global barriers per level thereby ensuring no out-of-order traversals occur between processes.
- An allreduce operation at the checks to see if the algorithm has finished
- The concurrency of this breadth first search parallel algorithm is naturally tied to the edge list and the traversal tasks that result from these traversing these edges.
- While many parallel algorithms could be recast as task dataflow parallelism, graph and knowledge management applications tend to be naturally expressed using this parallel model.

# Libraries

# Libraries

- Widely varying computational science disciplines use a significant amount of the available high performance computing resources.

XSEDE Allocations Summary -- 2015



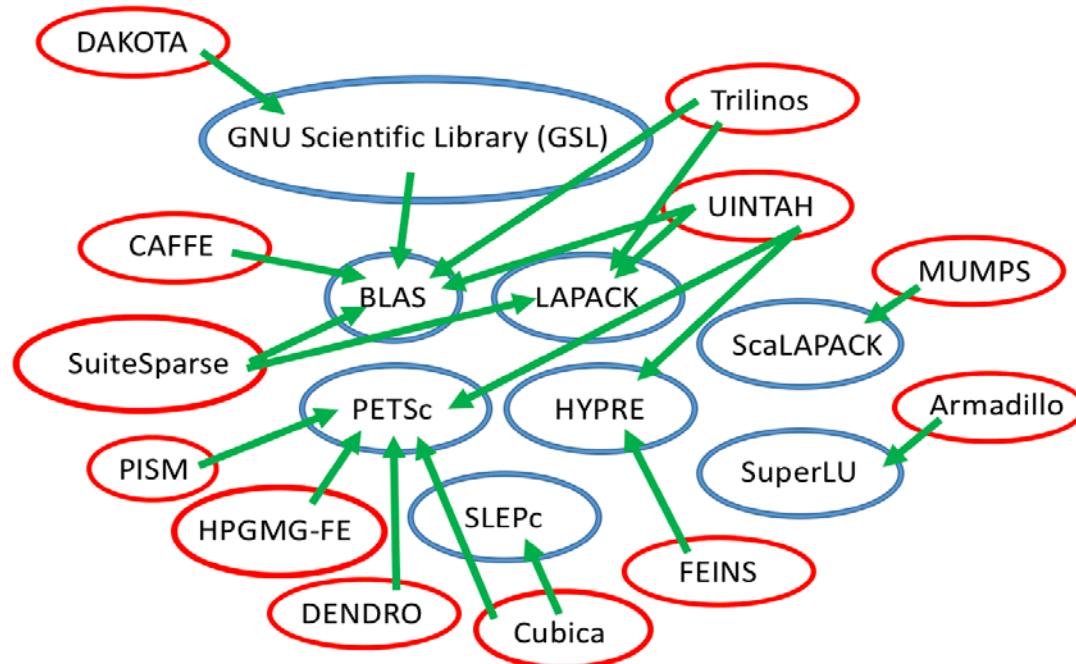
# Purpose of libraries

- Portability – tuned to perform across a wide range of architectures
- Performance – tuned over a decade or more for performance
- Code-reuse -- Avoid wasting developer time in re-developing software for a core operation shared across many research areas
- Enable vendors to tune implementations for emerging hardware
- Strongly support benchmarking efforts, sometimes even resulting in symbiotic relationships, e.g. BLAS and LINPACK.

# Widely Used HPC libraries and their Application Domain

Application Domain	Widely Used Libraries on High Performance Computing Systems
Linear Algebra	BLAS, LAPACK, ScalAPACK, GNU Scientific Library, SuperLU, PETSc, SLEPc, ELPA, Hypre
Partial Differential Equations	PETSc, Trilinos
Graph Algorithms	Boost Graph Library, Parallel Boost Graph Library
I/O	HDF5, Netcdf, Silo
Mesh Decomposition	METIS, ParMETIS
Visualization	VTK
Parallelization	Pthreads, MPI, Boost MPI
Signal Processing	FFTW
Performance Monitoring	PAPI, Vampir

## Examples of Library Dependencies in Common HPC Applications



Blue – core linear algebra libraries, Red – application frameworks dependent on core linear algebra libraries, Green -- dependencies

# BLAS

- BLAS level 1 provided a standard, machine independent application interface to vector routines optimized for specific computer architectures
- The BLAS idea is credited to Charles Lawson and Richard Hanson at the Jet Propulsion Laboratory in the 70's.



- Lawson, Hanson, F. Krogh, D.R. Kincaid and Jack Dongarra were responsible for the conception, design and implementation of the BLAS.
- Stands for Basic Linear Algebra Subprograms (the term “subprograms” is a Fortran term referring to functions and subroutines)
- BLAS development was conducted in conjunction with the LINPACK project undertaken by Argonne National Laboratory, and the two projects proved symbiotic.
- LINPACK was the first major package to use BLAS.

# CDC 7600

- BLAS Level 1 was motivated by the CDC 7600 with an instruction cache of 16 instructions
- The world's fastest computer from 1969– 1975 (up to 36 Mflops)
- Successor to the world's first "supercomputer", the CDC 6600
- Designed by Seymour Cray
- **Generally required hand compiled assembly code to achieve good performance.**



“You didn’t want to do that with a big piece of software, and so it focused the idea on identifying small pieces of software that are significant in the overall running time of the program. Of course in linear equation solving, it’s the classic case.” --Charles Lawson on why BLAS was created.

# BLAS

- The first BLAS routines were 10-15 routines and were just vector operations (inner products, norms, adding vectors, scalar multiplication, etc)

$$\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$$

- The original BLAS dealt only with one level of looping, essentially with vectors rather than matrices because the cache size on the CDC 7600 machines was so small that there wasn't any point in doing matrices.
- 1987 – Level 2 BLAS: Matrix-vector operations (about 10 years after Level 1 BLAS)

$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta \mathbf{y}$$

- 1989 – Level 3 BLAS: Matrix-matrix operations

$$C \leftarrow \alpha AB + \beta C$$

# BLAS

- Several implementations for different languages exist
  - Reference implementation (F77 and C-wrapper)  
<http://www.netlib.org/blas/>
  - ATLAS, highly optimized for particular processor architectures
  - A generic C++ template class library providing BLAS functionality: uBLAS  
<http://www.boost.org>
  - Several vendors provide libraries optimized for their architecture (AMD, HP, IBM, Intel, NEC, NViDIA, Sun)

# The core BLAS precision prefixes

Prefix	Description
s	Single precision (float), 4 bytes
d	Double precision (double), 8 bytes
c	Complex, (two floats), 8 bytes
z	Complex*16, (two doubles), 16 bytes

# BLAS Level 1 operations : Vector Rotations

Name	Description	Supported Precisions
<b>rotg</b>	Computes the parameters for a Givens rotation. That is, given scalars a and b, compute c and s so that $\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$ where $r = \sqrt{ a ^2 +  b ^2}$	s,d
<b>rot</b>	Applies the Givens rotation. That is, provided two vectors as input, x and y, each vector element is replaced as follows: $x_i = cx_i + sy_i$ $y_i = -sx_i + cy_i$ where c and s are the parameters for the Givens rotation (See rotg).	s,d
<b>rotmg</b>	Computes the $2 \times 2$ modified Givens rotation matrix $H = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}$ That is, given scaling factors $d_1$ and $d_2$ with Cartesian coordinates $(x_1, y_1)$ of an input vector, compute the modified Givens rotation matrix H such that $\begin{pmatrix} x_1 \\ 0 \end{pmatrix} = H \begin{pmatrix} x_1\sqrt{d_1} \\ y_1\sqrt{d_2} \end{pmatrix}$	s,d
<b>rotm</b>	Applies the modified Givens rotation. That is, provided two vectors, x and y, compute: $\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$ where $h_{ij}$ are the elements of the modified Givens rotation matrix (See rotmg).	s,d

# BLAS Level 1 operations : Vector Operations without a dot product

Name	Description	Supported Precisions
<b>swap</b>	Swaps vectors $x \leftrightarrow y$	s,d,c,z
<b>scal</b>	Scales a vector by a constant $y = \alpha y$	s,d,c,z,cs,zd
<b>copy</b>	Copies a vector $y = x$	s,d,c,z
<b>axpy</b>	Updates a vector $y = \alpha x + y$	s,d,c,z

# BLAS Level 1 operations : Vector Operations with a dot product

Name	Description	Supported Precisions
<b>dot</b>	Dot product $x^T y$	s,d,ds
<b>dotc</b>	Complex conjugate dot product $x^h y$	c,z
<b>dotu</b>	Complex dot product $x^T y$	c,z
<b>sdsdot</b>	Dot product plus a scalar $\alpha + x^T y$	sds

# BLAS Level 1 operations : Vector Norms

Name	Description	Supported Precisions
<b>nrm2</b>	Compute the 2-norm $\ x\ _2 = \sqrt{\sum  x_i ^2}$	s,d,sc,dz
<b>asum</b>	Compute the 1-norm $\ x\ _1 = \sum  x_i $	s,d,sc,dz
<b>i_amax</b>	Compute the $\infty$ -norm $\ x\ _\infty = \max( x_i )$	s,d,c,z

# Matrix Types supported in BLAS Level 2 and 3

Matrix Type	Description
General: ge, gb	General, nonsymmetric, possibly rectangular matrix
Symmetric: sy,sb,sp	Symmetric matrix. This is a special class of square matrix that is equal to its own transpose. So for matrix A with elements $a_{ij}$ , a symmetric matrix would have elements which satisfy $a_{ij} = a_{ji}$ .
Hermitian: he,hb,hp	Hermitian matrix. This is a special class of square matrix that is equal to its own Hermitian conjugate. So for matrix A with elements $a_{ml}$ , matrix A is Hermitian if all elements satisfy $a_{ml} = \overline{a_{lm}}$ where the overbar is the complex conjugate.
Triangular: tr,tb,tp	Triangular matrix. This is a special class of square matrices where all the entries above the diagonal are zero (lower triangular) or all the entries below the diagonal are zero (upper triangular).

# BLAS Level 2 and Level 3 operations

Name	Description
<b>mv</b>	Matrix-vector product
<b>sv</b>	Solve matrix (only for triangular matrices)
<b>mm</b>	Matrix-matrix product, $C = \alpha AB + \beta C$ where $A, B, C$ are matrices and $\alpha, \beta$ are scalars
<b>rk</b>	Rank-k update, $C = \alpha AA^T + \beta C$ where $A, C$ are matrices and $\alpha, \beta$ are scalars
<b>r2k</b>	Rank-2k update, $C = \alpha AB^T + \bar{\alpha} BA^T + \beta C$ where $A, B, C$ are matrices and $\alpha, \beta$ are scalars

# Example CBLAS DGEMM

```
void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                  const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                  const int K, const double alpha, const double *A,
                  const int lda, const double *B, const int ldb,
                  const double beta, double *C, const int ldc);
```

- *Order* indicates the storage layout as either row-major or column-major. This input is either *CblasRowMajor* or *CblasColMajor*.
- *TransA* indicates whether to transpose matrix A. This input is either *CblasNoTrans*, *CblasTrans*, or *CblasConjTrans* indicating no transpose, transpose, or complex conjugate transpose, respectively.
- *TransB* indicates whether to transpose matrix B. Acceptable options are the same as those listed for A.
- *M* indicates the number of rows in matrices A and C.
- *N* indicates the number of columns in matrices B and C.
- *K* indicates the number of columns in matrix A and the number of rows in matrix B. This is the shared index between matrices A and B.
- *alpha* is the scaling factor for  $A^*B$ .
- *A* is the pointer to matrix A data.
- *lda* is the size of the first dimension of matrix A.
- *B* is the pointer to matrix B data.
- *ldb* is the size of the first dimension of matrix B.
- *beta* is the scaling factor for matrix C.
- *C* is the pointer to matrix C data.
- *ldc* is the size of the first dimension of matrix C.

# Example CBLAS DGEMM

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cblas.h>
4
5 int main()
6 {
7     double *A, *B, *C;
8     int m = 3; // square matrix, number of rows and columns
9     int i,j;
10
11    A = (double *) malloc(m*m*sizeof(double));
12    B = (double *) malloc(m*m*sizeof(double));
13    C = (double *) malloc(m*m*sizeof(double));
14
15    // initialize the matrices
16    for (i=0;i<m;i++) {
17        for (j=0;j<m;j++) {
18            A[j + m*i] = j + m*i; // arbitrarily initialized
19            B[j + m*i] = 3.14*(j + m*i);
20            C[j + m*i] = 0.0;
21        }
22    }
23    double alpha = 1.0;
24    double beta = 0.0;
25
26    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
27                 m, m, m, alpha, A, m, B, m, beta, C, m);
28
29    for (i=0;i<m;i++) {
30        for (j=0;j<m;j++) {
31            printf(" C[%d][%d]=%g ",i,j,C[j+m*i]);
32        }
33        printf("\n");
34    }
35
36
37    free(A);
38    free(B);
39    free(C);
40    return 0;
41 }
```

$$\begin{pmatrix} 47.1 & 56.52 & 65.94 \\ 131.88 & 169.56 & 207.24 \\ 216.66 & 282.6 & 348.54 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 0 & 3.14 & 6.28 \\ 9.42 & 12.56 & 15.7 \\ 18.84 & 21.98 & 25.12 \end{pmatrix}$$

# LAPACK Driver Routines

Driver Name	Description
SV	Solver for system of linear equations: $Ax = b$
LS, LSY, LSS, LSD	Solver for linear least squares problems: minimize $x$ in $\ b - Ax\ _2$ where $A$ is not necessarily a square matrix, generally with more rows than columns as would occur in an overdetermined system of linear equations
LSE	Linear equality-constrained least squares problems: minimize $x$ in $\ c - Ax\ _2$ subject to the constraint that $Bx = d$ where $A$ is an $m \times n$ matrix, $c$ is a vector of size $m$ , $B$ is a $p \times n$ matrix, and $d$ is a vector of size $p$ , where $p \leq n \leq m + p$ .
GLM	General linear model problems: minimize $x$ in $\ y\ _2$ subject to the constraint that $d = Ax + By$ where $A$ is an $m \times n$ matrix, $B$ is a $n \times p$ matrix, $d$ is a vector of size $n$ , and $m \leq n \leq m + p$ .
EV, EVD, EVR	Symmetric eigenvalue problems: Find eigenvalues $\lambda$ and eigenvectors $k$ where $Ak = \lambda k$ for a symmetric matrix $A$ .
ES	Nonsymmetric eigenvalue problems: Find eigenvalues $\lambda$ and eigenvectors $k$ where $Ak = \lambda k$ for nonsymmetric matrix $A$ .
SVD, SDD	Compute the singular value decomposition of $m \times n$ matrix $A$ : $A = UDV^T$ where matrices $U$ and $V$ are orthogonal and $D$ is a diagonal real matrix of size $m \times n$ containing the singular values of matrix $A$ .

# LAPACK Example: DGESV

```
lapack_int LAPACKE_dgesv( int matrix_layout, lapack_int n, lapack_int nrhs,  
                           double* a, lapack_int lda, lapack_int* ipiv,  
                           double* b, lapack_int ldb );
```

- *matrix\_layout* specifies the whether the matrix is specified in row-major or column-major form. Acceptable inputs are either LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR.
- *n* indicates the size of the square matrix.
- *nrhs* indicates the number of right hand side vectors on which to perform the solve. dgesv can solve multiple right hand sides in each call.
- *a* is the matrix.
- *lda* is the size of the first dimension of the matrix.
- *ipiv* is a vector of size *n* containing the pivot points.
- *b* is the right hand side vector.
- *ldb* is the size of the first dimension of the right hand side vector.

# LAPACK Example: DGESV

```
1 #include <stdio.h>
2 #include <lapacke.h>
3
4 int main (int argc, const char * argv[])
5 {
6     double A[3][3] = {1,3,2,4,1,9,5,7,2};
7     double b[3] = {-1,-1,1};
8     lapack_int ipiv[3];
9     lapack_int info,m,lda,ldb,nrhs;
10    int i,j;
11
12    m = 3;
13    nrhs = 1;
14    lda = 3;
15    ldb = 1;
16
17    // Solve the linear system
18    info = LAPACKE_dgesv(LAPACK_ROW_MAJOR,m,nrhs,*A,lda,ipiv,b,ldb);
19
20    // check for singularity
21    if (info > 0 ) {
22        printf(" U(%d,%d) is zero! A is singular\n",info,info);
23        return 0;
24    }
25
26    // print the answer
27    for (i=0;i<m;i++) {
28        printf(" b[%i] = %g\n",i,b[i]);
29    }
30
31    printf( "\n" );
32    return 0;
33 }
```

Solve for x:

$$\begin{pmatrix} 1 & 3 & 2 \\ 4 & 1 & 9 \\ 5 & 7 & 2 \end{pmatrix} x = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$$

# GSL BLAS DGEMM Example

```
1 #include <stdio.h>
2 #include <gsl/gsl blas.h>
3
4 int main (void) {
5     double a[] = { 0,1,2,
6                     3,4,5,
7                     6,7,8 };
8
9     double b[] = { 0,      3.14, 6.28,
10                  9.42, 12.56,15.7,
11                  18.84,21.98,25.12 };
12
13    double c[] = { 0.00, 0.00, 0.00,
14                  0.00, 0.00, 0.00,
15                  0.00, 0.00, 0.00 };
16
17    gsl_matrix_view A = gsl_matrix_view_array(a, 3, 3);
18    gsl_matrix_view B = gsl_matrix_view_array(b, 3, 3);
19    gsl_matrix_view C = gsl_matrix_view_array(c, 3, 3);
20
21 // Compute C = A B
22
23    gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
24                      1.0, &A.matrix, &B.matrix,
25                      0.0, &C.matrix);
26
27    printf (" %g, %g, %g\n", c[0], c[1],c[2]);
28    printf (" %g, %g, %g\n", c[3], c[4],c[5]);
29    printf (" %g, %g, %g\n", c[6], c[7],c[8]);
30
31    return 0;
32 }
```

# PETSc

A small sample of distributed vector operations in PETSc

Vector Function Name	Description
<b>VecAXPY(Vec y, PetscScalar alpha, Vec x)</b>	$y = \alpha x + y$
<b>VecAYPX(Vec y, PetscScalar alpha, Vec x)</b>	$y = x + \alpha y$
<b>VecPointwiseMult(Vec w, Vec x, Vec y)</b>	$w_i = x_i * y_i$
<b>VecMax(Vec x, PetscInt *p, PetscReal *r)</b>	Returns the max value, $r = \max(x_i)$ , and its location
<b>VecCopy(Vec x, Vec y)</b>	$y = x$
<b>VecShift(Vec x, PetscScalar s)</b>	$x_i = s + x_i$
<b>VecScale(Vec x, PetscScalar alpha)</b>	$x = \alpha x$

# Operating Systems

# Operating System

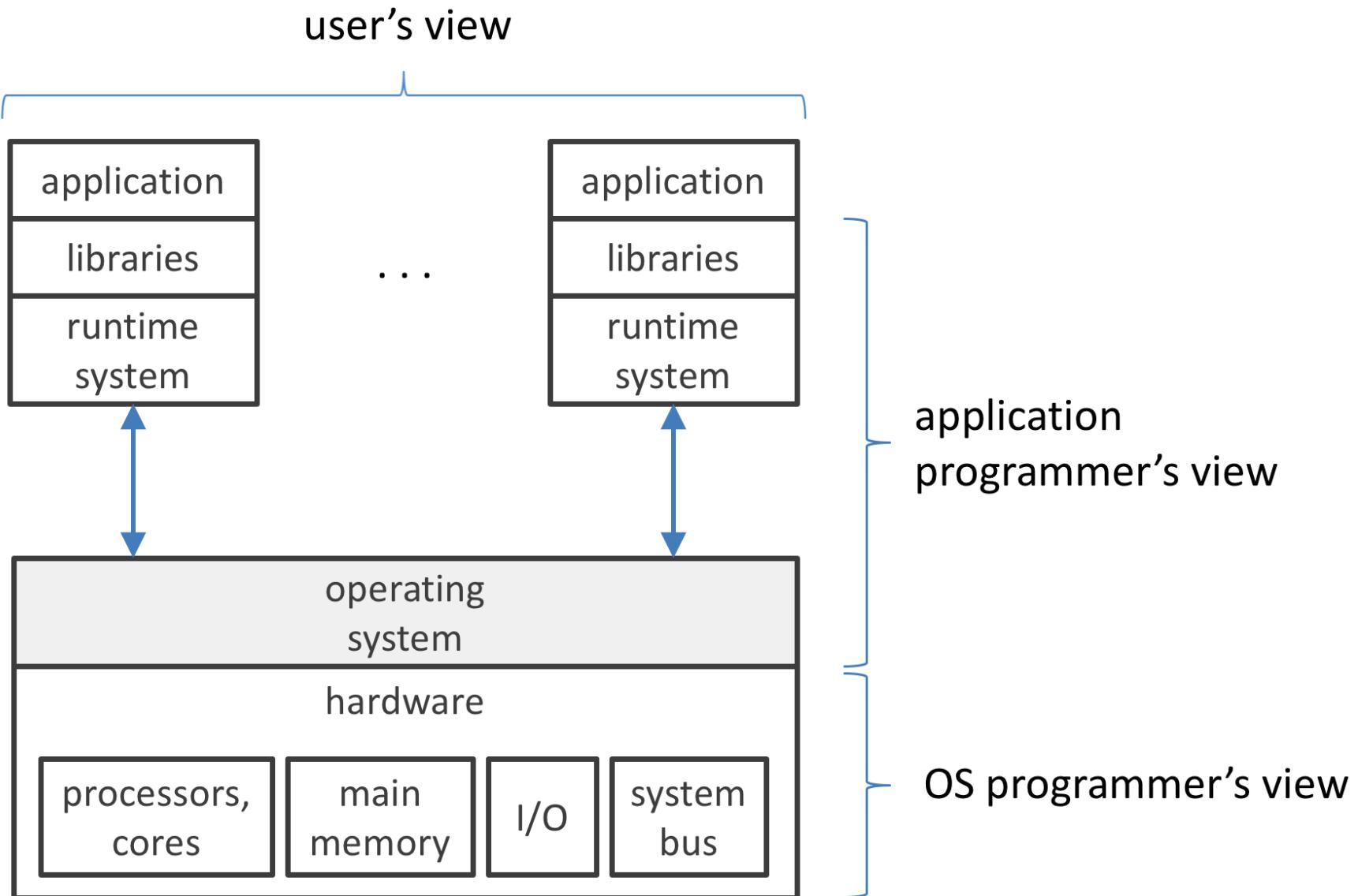
- What is an Operating System?
  - A program that **controls the execution of application programs**
  - An interface between applications and hardware
  - A program that manages hardware resources and task scheduling
- Primary functionality
  - Exploits the hardware resources of one or more processors
  - Provides a set of services to system users
  - Manages secondary memory and I/O devices
- Objectives
  - Convenience: Makes the computer more convenient to use
  - Efficiency: Allows computer system resources to be used in an efficient manner
  - Ability to evolve: Permit effective development, testing, and introduction of new system functions without interfering with service

# Linux Distributions

Alphanet Linux	Embedix	Linux by Linux	Platinum Linux	Vine Linux
Alzza Linux	Enoch	Linux GT Server Edition	Power Linux	White Dwarf Linux
Andrew Linux	Eonova Linux	Linux Mandrake	Progeny Debian	Whole Linux
Apokalypse	ESware	Linux MX	Project Freesco	WinLinux 2000
Armed Linux	Etlinux	LinuxOne	Prosa Debian	WorkGroup Solutions
ASPLinux	Eurielec Linux	LinuxPPC	Pygmy Linux	Linux Pro Plus
Bad Penguin	FinnixFloppi	Gentoo	Red Flag Linux	Xdenu
Bastille Linux	Linux	LinuxPPP	Red Hat Linux	Xpresso Linux 2000
Best Linux	Gentus Linux	LinuxSIS	Redmond Linux	XTeam Linux
BlackCat Linux	Green Frog Linux	LinuxWare	Rock Linux	Yellow Dog Linux
Blue Linux	Halloween Linux	Linux-YeS	RT-Linux	Yggdrasil Linux
Bluecat Linux	Hard Hat Linux	LNX System	Scrudge Ware	ZiiF Linux
BluePoint Linux	HispaFuentes	Lunet	Secure Linux	ZipHam
Brutalware	HVLinux	LuteLinux	Skygate Linux	ZipSlack
Caldera OpenLinux	Icepack	LST	Slacknet Linux	
Cclinux	Immunix	Mastodon	Slackware	
ChainSaw Linux	OSIndependence	MaxOS&trade;	Slinux	
CLECIIeNUX	InfoMagick Workgroup	MIZI Linux OS	SOT Linux	
Conectiva	Server	MkLinux	Spiro	
CoolLinux	Ivrix	MNIS Linux	Stampede Linux	
Coyote Linux	ix86 Linux	MicroLinux	Storm Linux	
Corel	JBLinux	Monkey Linux	S.u.SE	
COX-Linux	Jurix Linux	NeoLinux	Thin Linux	
Darkstar Linux	Kondara	Newlix OfficeServer	TINY Linux	
Debian Definite	Krud	NoMad Linux	Trinux	
Linux	KW Linux	Ocularis	Trustix Secure Linux	
deepLINUX	KSI Linux	Open Kernel Linux	TurboLinux	
Delix	L13Plus	Open Share Linux	Turquaz	
Dlite (Debian Lite)	Laser5	OS2000	UltraPenguin	
DragonLinux	Leetnux	Peanut Linux	Ute-Linux	
Eagle Linux M68K	Lightening	PhatLINUX	VA-enhanced RedHat Linux	
easyLinux	Linpus Linux	PingOO	VectorLinux	
Elfstone Linux	Linux Antarctica	Plamo Linux	Vedova Linux	

# Services Provided by the OS

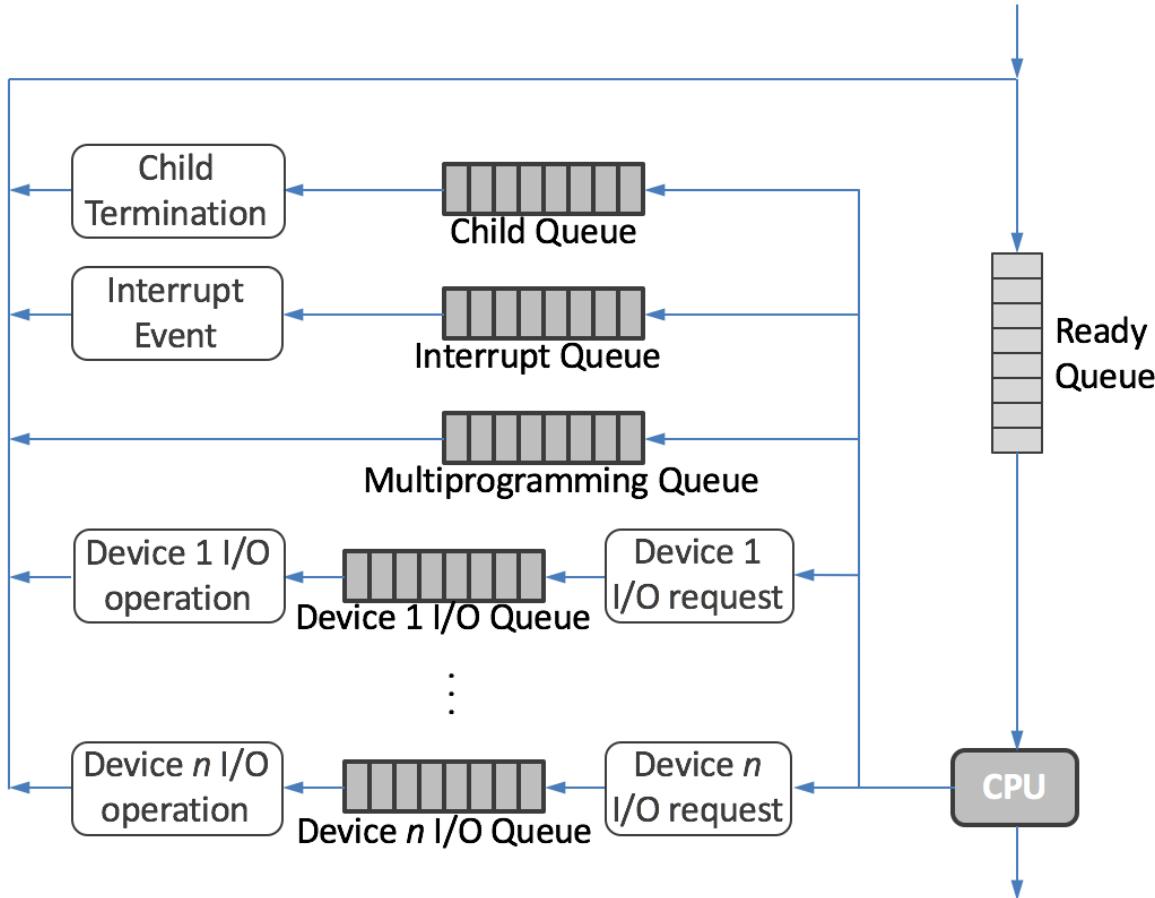
- Program development
  - Editors and debuggers
- Program execution
- Access to I/O devices
- Controlled access to files
- System access
- Protection
- Error detection and response
  - Internal and external hardware errors
  - Software errors
  - Operating system cannot grant request of application
- Accounting



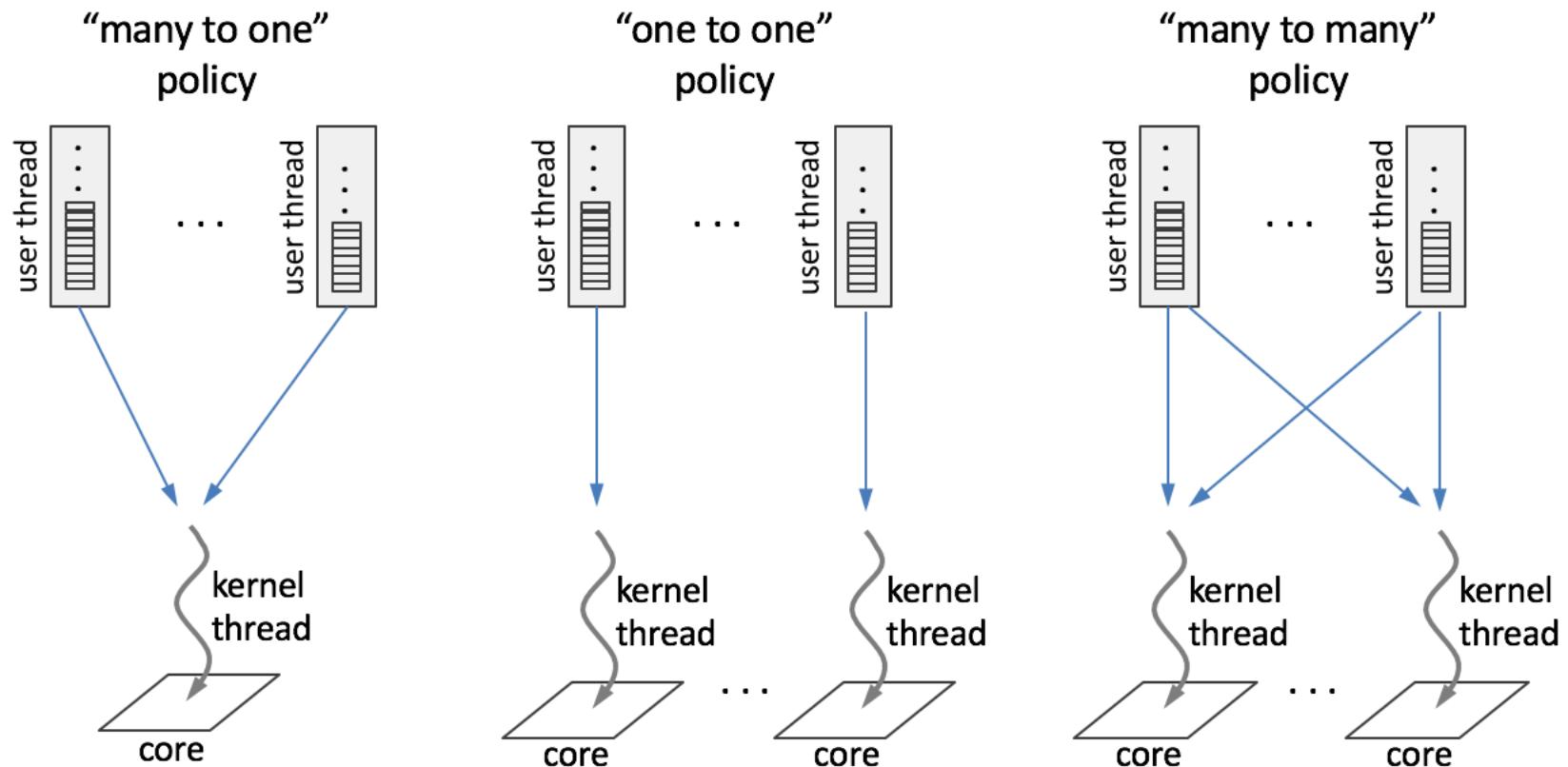
# Resources Managed by the OS

- Processor
- Main Memory
  - volatile
  - referred to as real memory or primary memory
- I/O modules
  - secondary memory devices
  - communications equipment
  - terminals
- System bus
  - communication among processors, memory, and I/O modules

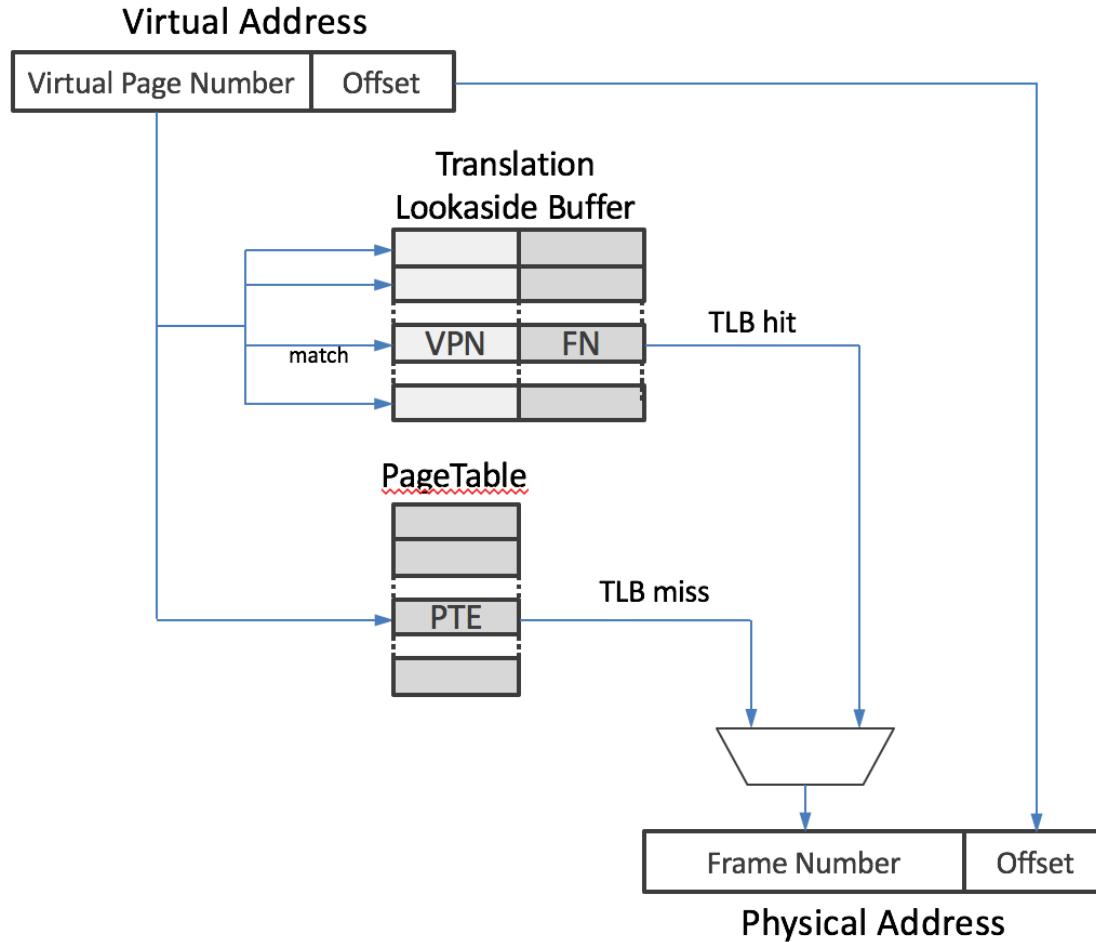
At the heart of the process management services supported by the OS is the cross-cutting functionality of process scheduling: the determination of what processes are given the necessary physical resources to run and when they are allocated.



The job queue holds all processes, whatever their states, and any new process entering the system is put in the job queue by the OS. A representative job queue may include a ready queue, child queue, interrupt queue, multiprogramming queue, and I/O queues.



The runtime system allocates the kernel threads made available to it to the user threads for which it is responsible in several different ways.

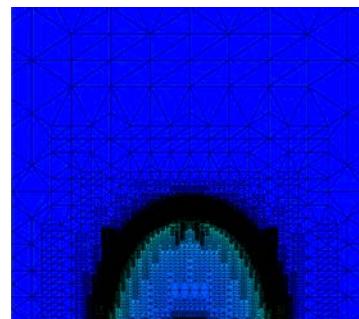
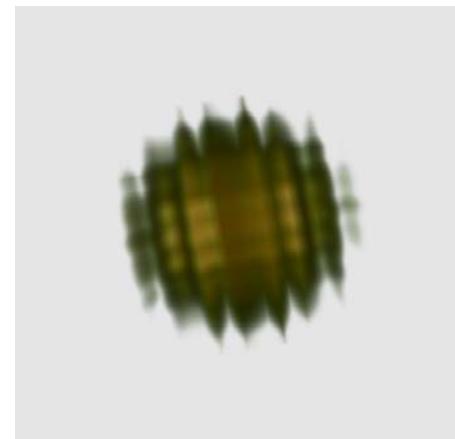
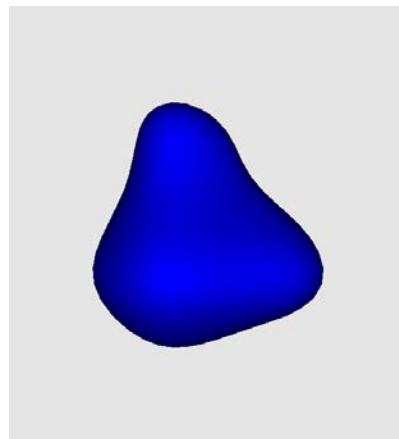
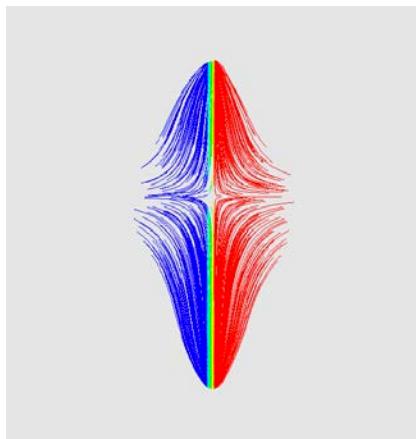


The translation lookaside buffer (TLB) is a special-purpose cache that operates to provide high-speed mapping of virtual page numbers to main memory frame numbers for recently used stored data.

# Visualization

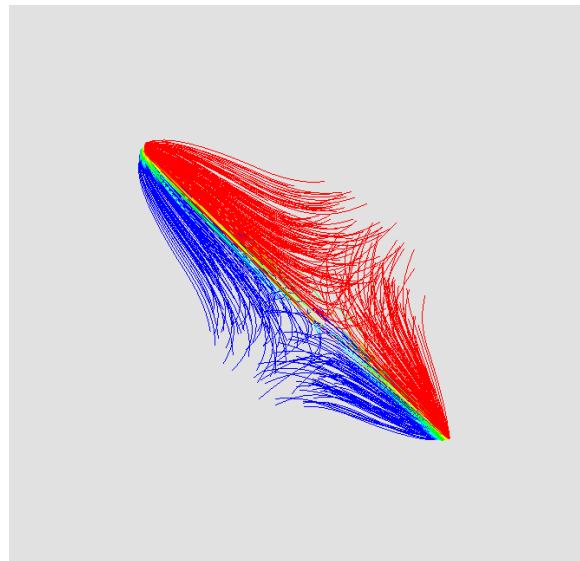
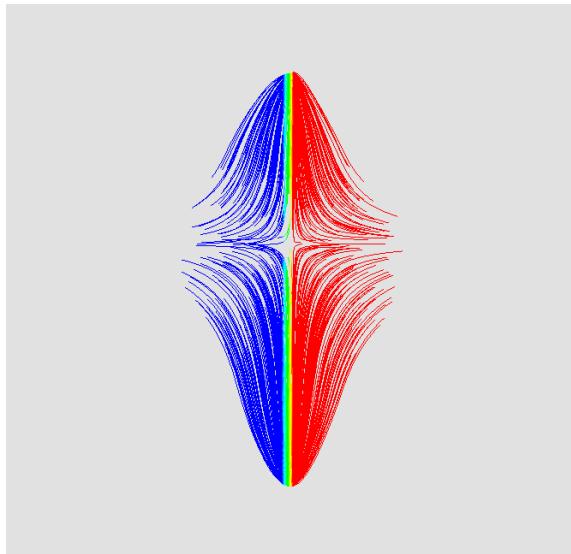
# Visualization Concepts

- Many scientific visualizations incorporate at least one of some foundational visualization concepts: streamlines, isosurfaces, volume rendering by ray tracing, and mesh tessellations.



# Streamlines

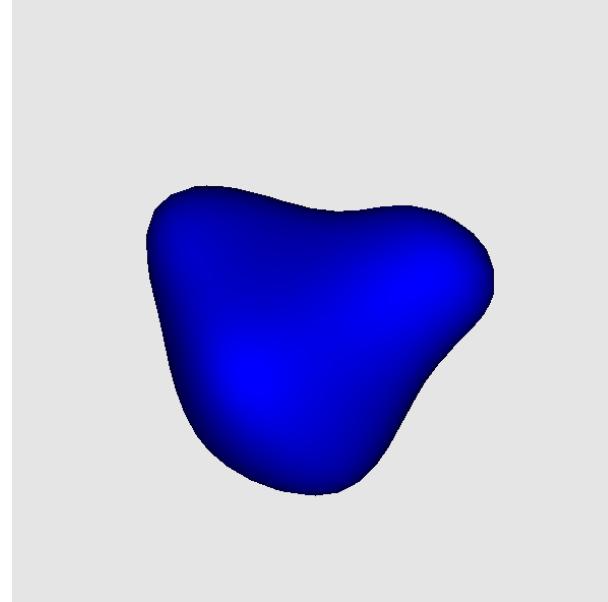
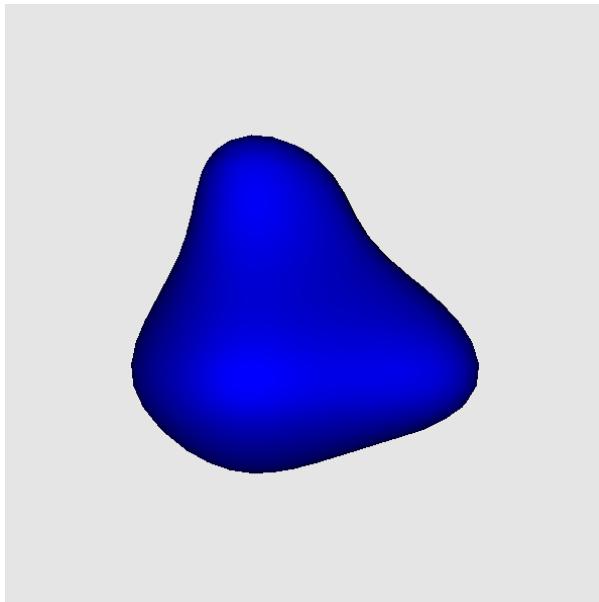
- Streamlines take a vector field as input and show curves that are tangent to the vector field



. Streamline example using the gradient of the function  $f(x,y,z) = 2550 \sin(100x) \sin(30y) \cos(40z)$  as input. Two different 3-D views are provided.

# Isosurfaces

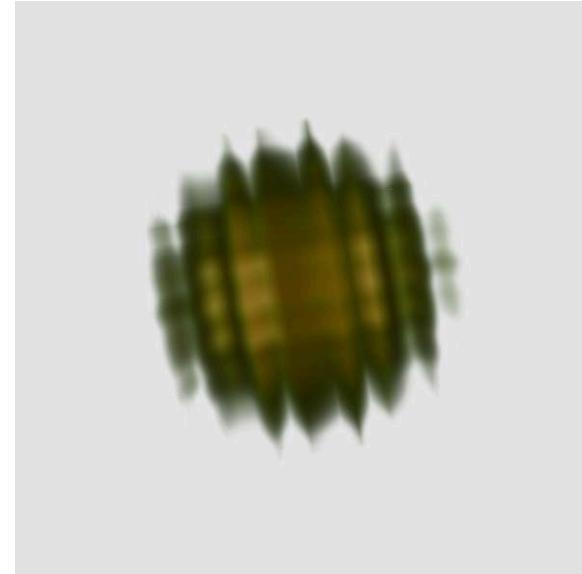
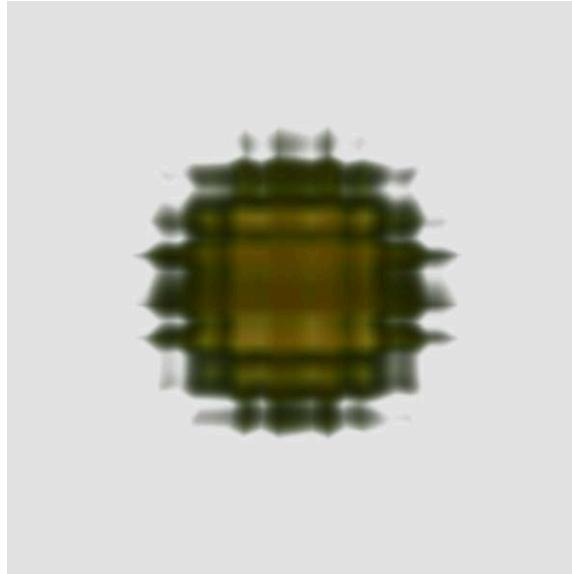
- Isosurfaces are surfaces which connect data points which have the same value.



. Isosurface example of the function  $f(x,y,z) = 2550 \sin(10x) \sin(10y) \cos(10z)$  as input with the isosurface value set at 200. Two different 3-D views are provided.

# Volume Rendering

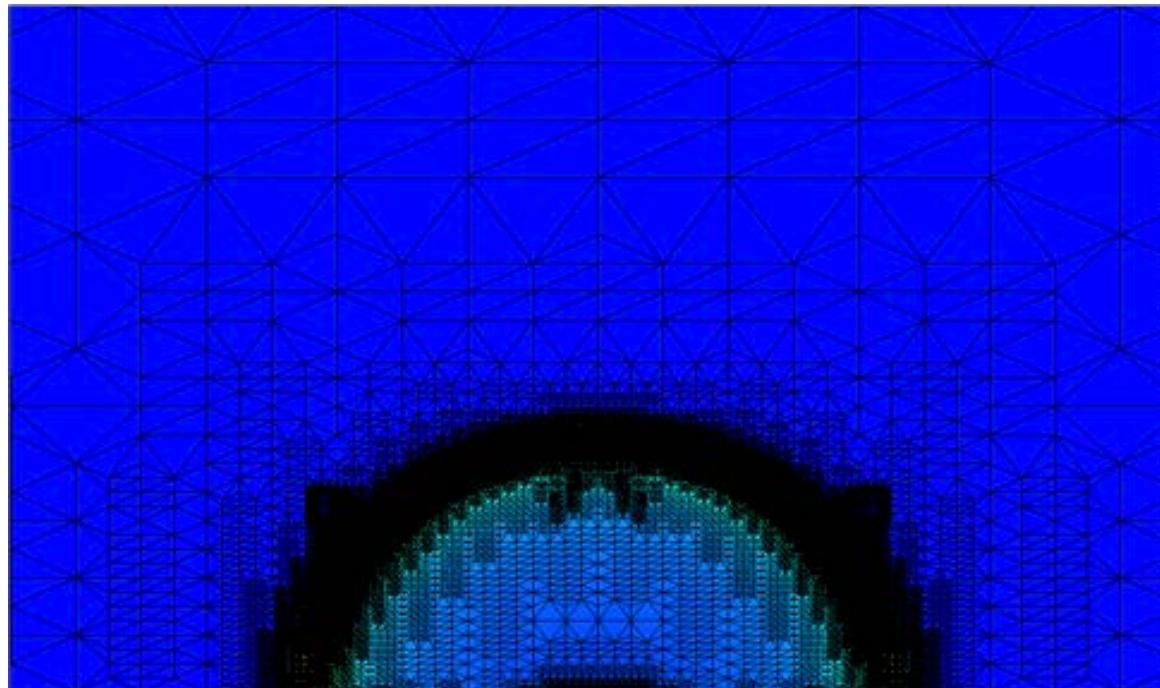
- Volume rendering by ray tracing casts rays through the data volume and sample the volume through which the ray passes.



*Example of low resolution volume rendering of the function  $f(x,y,z) = 2550 \sin(50x) \sin(50y) \cos(50z)$ . The color and opacity map were chosen arbitrarily. Two different 3-D views are provided.*

# Mesh Tessellation

- Mesh tessellations visualize data points and their connectivities to other data points using polygons.

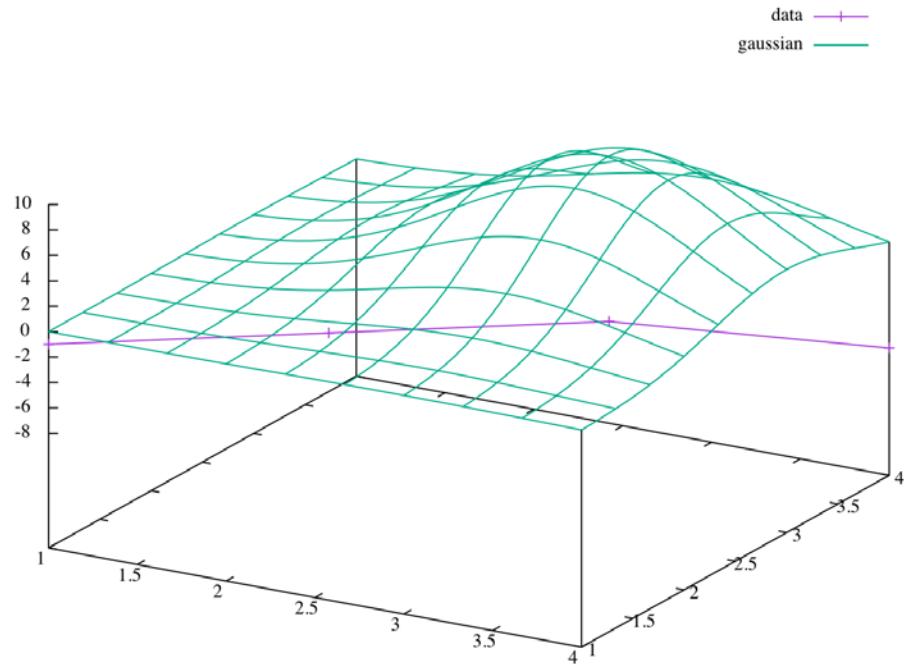


# Common Tools for Scientific Visualization

- Gnuplot is a simple command line visualization tool for 2-D and 3-D plots.
- Matplotlib is a Python based visualization tool with easy integration to other libraries with Python bindings.
- VTK is an open source collection of visualization algorithms for creating application specific visualization solutions.
- ParaView and VisIt are turnkey visualization solutions incorporating VTK algorithms but providing a GUI and scripting interface for visualization.

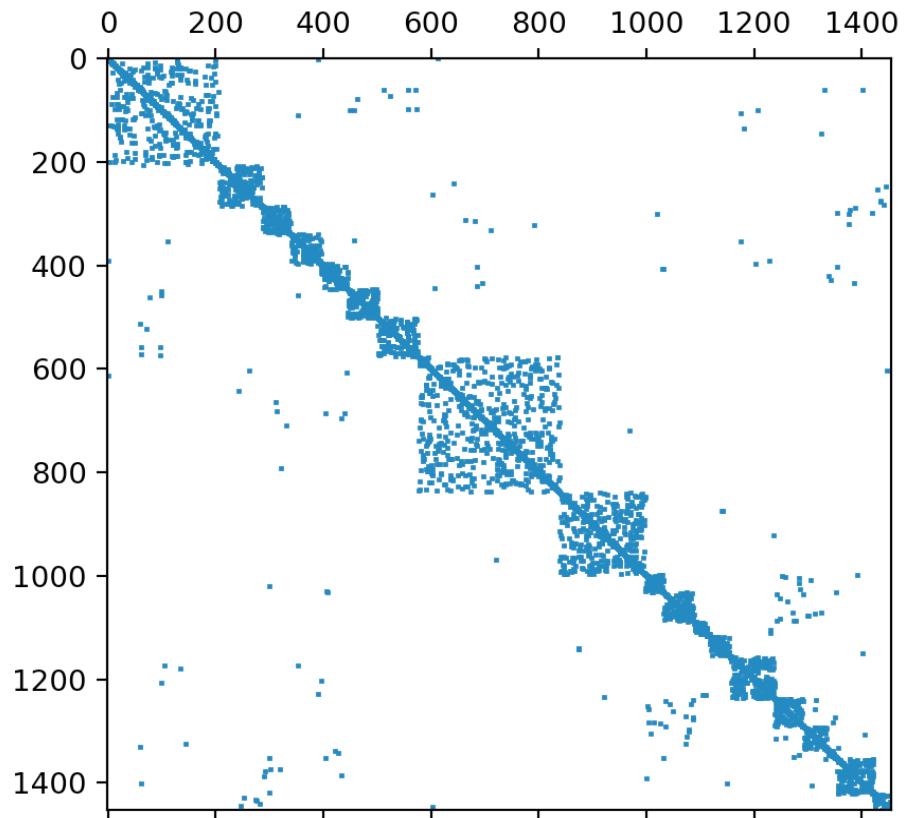
# Gnuplot

```
plot "gnu_example.dat" with linespoints title "data", 10*exp(-(x-3)**2-(y-3)**2)
title "gaussian"
```

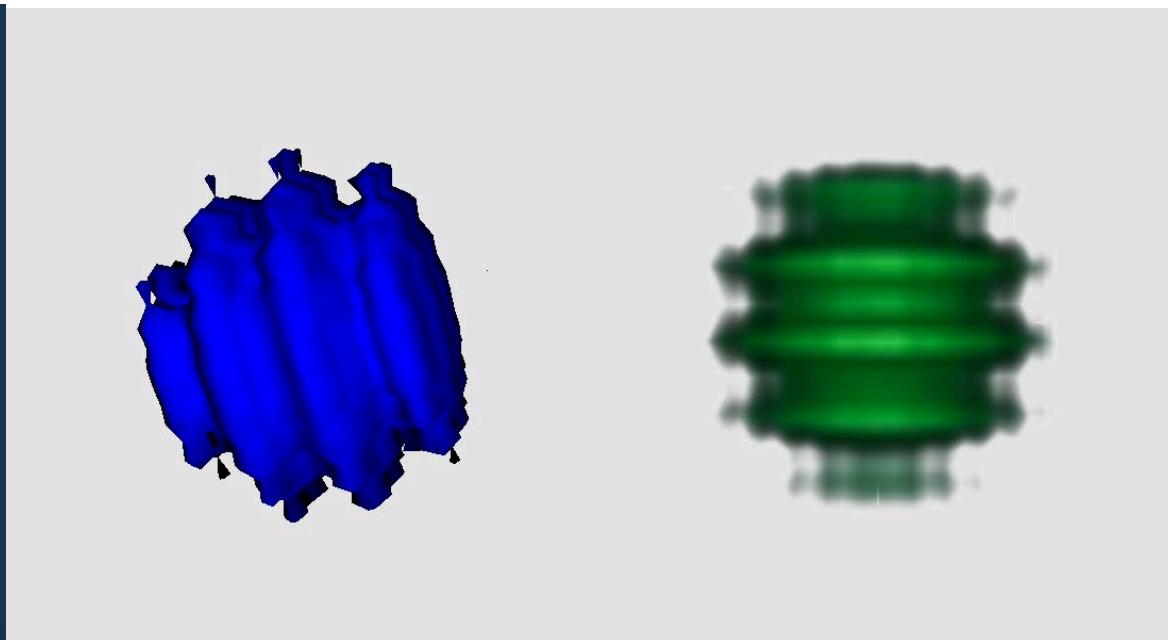
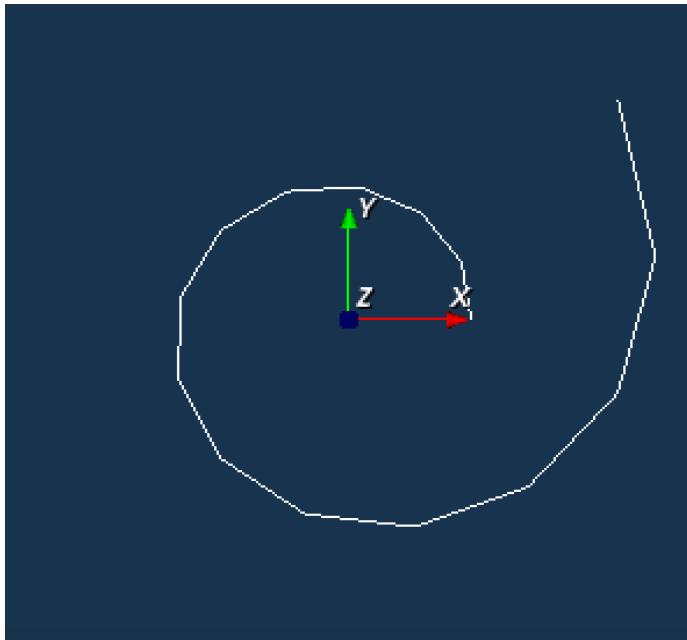


# Matplotlib

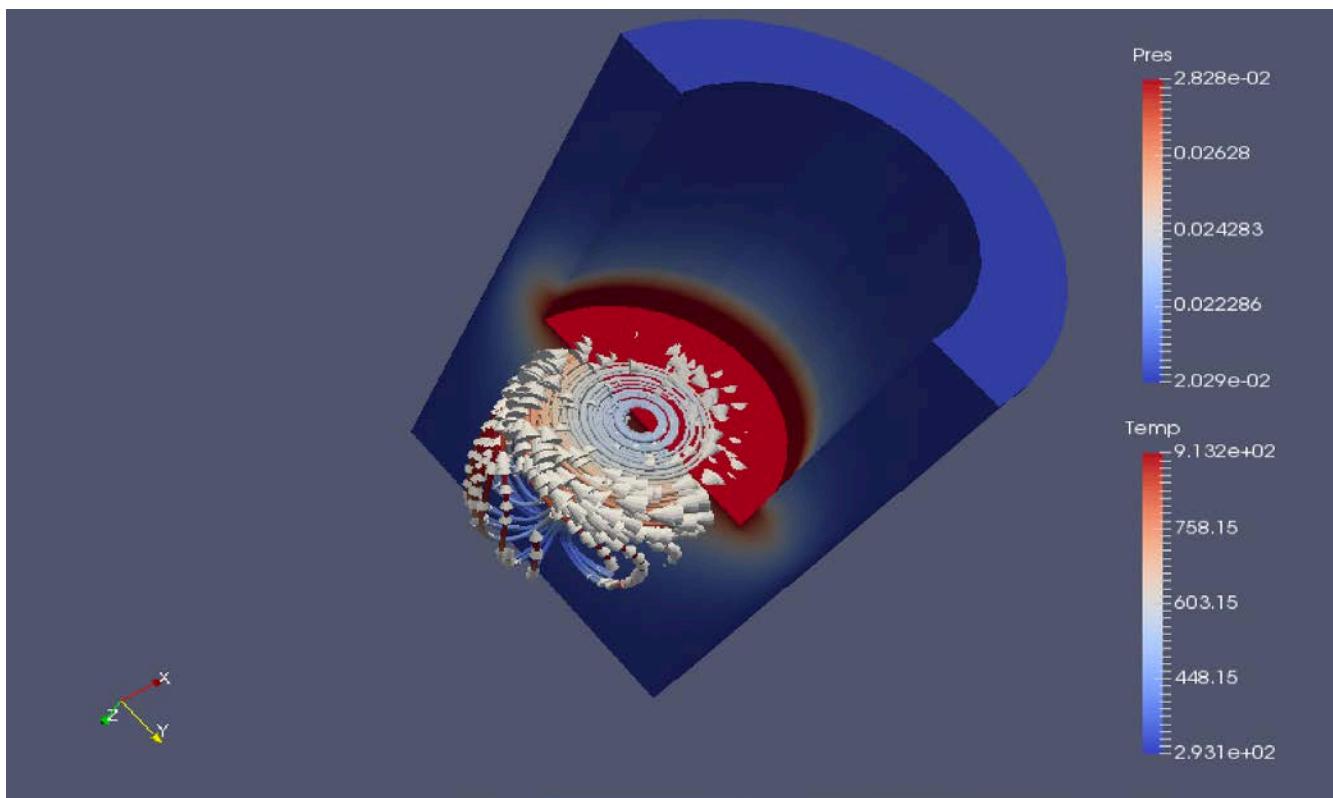
```
1 import scipy.io as sio
2 from matplotlib.pyplot import figure, show
3 import numpy
4
5 A = sio.mmread("bcspwr06.mtx");
6
7 fig = figure()
8 ax1 = fig.add_subplot(111)
9
10 ax1.spy(A,markersize=1)
11 show()
```



# VTK Examples



# Paraview Example



# VisIt Example

DB: mhd.30000.pdb

Cycle: 0

Pseudocolor

Var: rho

0.1788

0.1341

**3 . 30**

0.08939

0.04470

6.312e-06

Max: 0.1788

**3 . 20**

Min: 6.312e-06

*Y Axis*

**3 . 10**

**3 . 00**

**2 . 90**

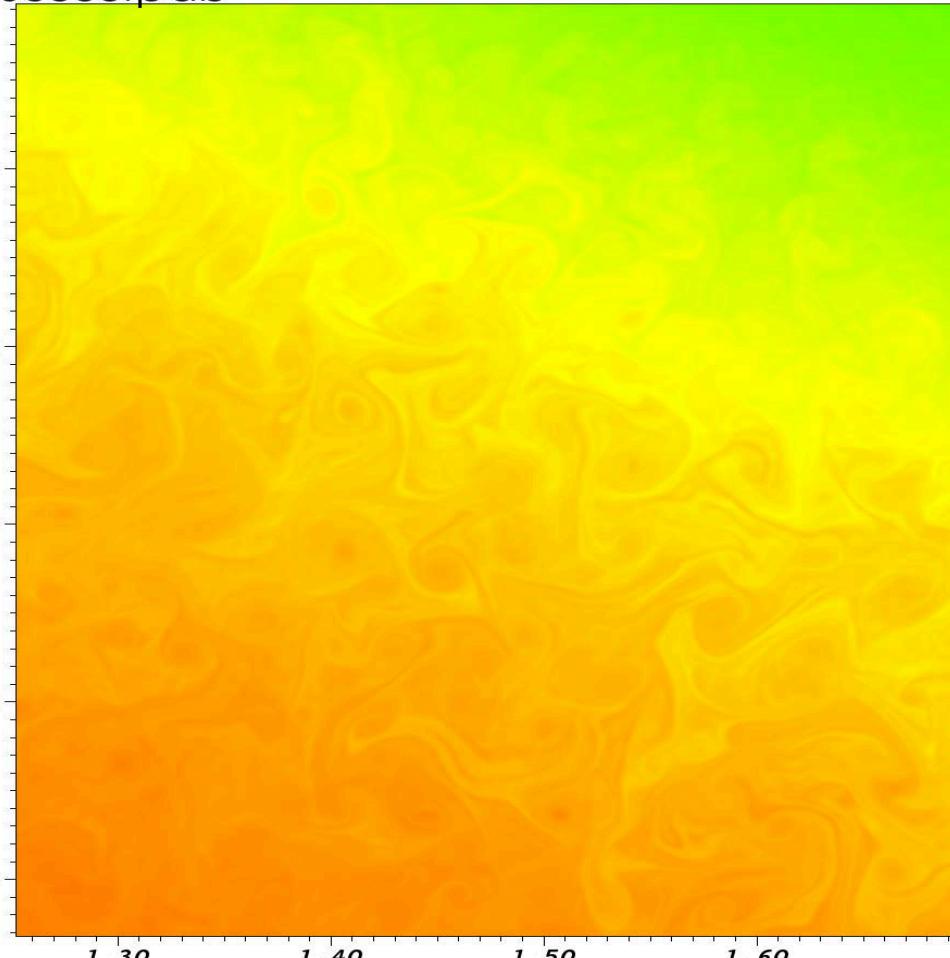
1 . 30

1 . 40

1 . 50

1 . 60

*X Axis*



# Performance Monitoring

# Rationale and Issues

- Performance inefficiencies are multiplied on a parallel computer
- The purpose of performance monitoring:
  - Elimination of performance degradation sources
  - Providing acceptable utilization of hardware resources
  - Verification that optimization yields the expected results (sanity check)
- Measurements impact program execution
  - The more invasive the instrumentation, the bigger the discrepancies
  - Program execution flow may be impacted in extreme cases
  - Overheads may be reduced through sampling
  - Many CPUs provide hardware support enabling event counting with extremely low overheads

# Select Proprietary Tools

- Intel VTune Amplifier
  - Targets multithreaded execution on the x86 line of CPUs and Xeon Phi
  - Profiling, hotspot analysis, memory usage and storage accesses, Flops and FPU utilization measurement, offload tracing via OpenCL
  - Integrated with Parallel Studio XE and Microsoft Visual Studio
  - Supported languages: C, C++, C#, Fortran, Java, Go, Python, assembly
- AMD CodeXL
  - Supports x86-compatible CPUs as well as AMD GPUs and APUs
  - Time based profiling on CPUs, event based profiling and instruction based sampling on CPUs and APUs, real-time power profiling
  - Available as a standalone version and as an extension to MS Visual Studio
- Nvidia visual profiler from CUDA Toolkit (nvvp)
  - Trace based tool
  - Execution timeline decomposed into individual threads and workload phases
  - Captures memory usage, power consumption, clock speed, thermal state
  - Monitors Pthreads on host CPU as well as OpenACC applications via PGI compiler
  - Available for Linux, OS X, and Windows

# Time Measurement: Command Line Tools

- “Date” utility
  - Single second accuracy, useful for coarse time stamps

```
> date  
Sun, Feb 05, 2017  6:17:33 PM
```

- “Time” utility
  - Captures actual duration of command execution as well as user and system time in seconds (bash shell built-in)
  - System version provides CPU utilization, memory usage, I/O, and page fault counts in addition to data output by bash command

```
> /usr/bin/time sleep 2  
0.00user 0.04system 0:02.04elapsed 2%CPU (0avgtext+0avgdata  
422144maxresident)k  
0inputs+0outputs (1737major+0minor)pagefaults 0swaps
```

# Time Measurement: API

- POSIX high-resolution clocks

- A number of clocks of different properties provided by the OS
- Monotonic clocks (`CLOCK_MONOTONIC` or `CLOCK_MONOTONIC_RAW`) most useful for program monitoring

```
#include <time.h>
int clock_gettime(clockid_t id, struct timespec *tsp);
```

- Struct `timespec` contains fields `tv_sec` and `tv_nsec` storing the number of full seconds and nanoseconds (respectively) elapsed from certain fixed point in the past
- The actual clock resolution may be retrieved by

```
int clock_getres(clockid_t id, struct timespec *tsp);
```

# Example: Matrix-Vector Multiply

```
001 #include <stdio.h>
002 #include <stdlib.h>
003 #include <cblas.h>
004 #include <time.h>
005
006 void init(int n, double **m, double **v, double *p, int trans) {
007     *m = calloc(n*n, sizeof(double));
008     *v = calloc(n, sizeof(double));
009     *p = calloc(n, sizeof(double));
010     for (int i = 0; i < n; i++) {
011         (*v)[i] = (i & 1)? -1.0: 1.0;
012         if (trans) for (int j = 0; j <= i; j++) (*m)[j*n+i] = 1.0;
013         else for (int j = 0; j <= i; j++) (*m)[i*n+j] = 1.0;
014     }
015 }
016
017 void mult(int size, double *m, double *v, double *p, int trans) {
018     int stride = trans? size: 1;
019     for (int i = 0; i < size; i++) {
020         int mi = trans? i: i*size;
021         p[i] = cblas_ddot(size, m+mi, stride, v, 1);
022     }
023 }
```

# Example: Matrix-Vector Multiply (cont.)

```
025 double sec(struct timespec *ts) {
026     return ts->tv_sec+1e-9*ts->tv_nsec;
027 }
028
029 int main(int argc, char **argv) {
030     struct timespec t0, t1, t2, t3, t4;
031     clock_gettime(CLOCK_MONOTONIC, &t0);
032     int n = 1000, trans = 0;
033     if (argc > 1) n = strtol(argv[1], NULL, 10);
034     if (argc > 2) trans = (argv[2][0] == 't');
035
036     double *m, *v, *p;
037     clock_gettime(CLOCK_MONOTONIC, &t1);
038     init(n, &m, &v, &p, trans);
039     clock_gettime(CLOCK_MONOTONIC, &t2);
040     mult(n, m, v, p, trans);
041     clock_gettime(CLOCK_MONOTONIC, &t3);
042     double s = cblas_dasum(n, p, 1);
043     clock_gettime(CLOCK_MONOTONIC, &t4);
044     printf("Size %d; abs. sum: %f (expected: %d)\n", n, s, (n+1)/2);
045     printf("Timings:\n program: %f s\n", sec(&t4)-sec(&t0));
046     printf("    init: %f s\n    mult: %f s\n    sum: %f s\n",
047           sec(&t2)-sec(&t1), sec(&t3)-sec(&t2), sec(&t4)-sec(&t3));
048     return 0;
049 }
```

# Example: Matrix-Vector Multiply Output

```
> ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Timings:
  program: 1.148853 s
    init: 0.572537 s
    mult: 0.576276 s
    sum: 0.000037 s
```

20000x20000 matrix, row-major mode

```
> ./mvmult 20000 t
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Timings:
  program: 12.126625 s
    init: 4.343727 s
    mult: 7.782852 s
    sum: 0.000043 s
```

20000x20000 matrix, column-major mode

# Application Profiling

- Identifies potential performance issues
- Hotspots
  - Parts of code the program spends most time executing
- Bottlenecks
  - Hotspots that have adverse impact on program execution
  - Optimizations may move them to other parts of code
- Instrumentation: program modification permitting collection of performance data
  - Manual (performed by the programmer)
  - Automated (performed by the compiler, library or external tool)
- Profiling data may be collected in user space, kernel space or both

# Common Profiling Metrics

- Memory related
  - Total memory, physical memory, shared memory
  - Size of program heap, stack or text segment
- I/O related
  - Number of input or output operations
  - Total amount of data transferred (read and write)
  - Data bandwidth (peak, average)
  - Number of files opened
- Communication
  - Number and size of messages sent or received
  - Achieved latency and bandwidth
  - Further categorization by network, endpoint type, and protocol

# Main Performance Classes

- Compute (or CPU) bound
  - Execution time is dominated by processor speed
- Memory bound
  - Duration of execution determined by amount of memory used
- I/O bound
  - The most significant fraction of program time is spent performing input/output operations

# gperftools

- Originally Google Performance Tools
- Currently community maintained
- Includes statistical profiling tool, *gprof*
- Additional tools accompanying thread-caching malloc library (tcmalloc)
  - Memory leak detection
  - Dynamic memory allocation profiling

# Gperf Example

- Compilation and linking

```
> gcc -O2 -ggdb mvmult.c -o mvmult -lcblas -lprofiler
```

- Invocation

```
> env CPUPROFILE=mvmult.prof ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
PROFILE: interrupts/evictions/bytes = 115/0/376
```

- Display of results

```
> pprof --text mvmult mvmult.prof
Using local file mvmult.
Using local file mvmult.prof.
Total: 115 samples
      58  50.4%  50.4%          58  50.4% ddot_
      57  49.6% 100.0%          57  49.6% init
        0   0.0% 100.0%          57  49.6% 0x00007f2c9485e00f
```

# Memory Monitoring Examples

- Leak detection

```
> gcc -O2 mvmult.c -o mvmult -lcbblas -ltcmalloc
> env HEAPCHECK=normal ./mvmult 20000
WARNING: Perftools heap leak checker is active -- Performance may suffer
tcmalloc: large alloc 3200000000 bytes == 0xe9e000 @ 0x7f887688eae7
0x4009b1 0x400b95
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Have memory regions w/o callers: might report false leaks
Leak check main detected leaks of 3200160000 bytes in 2 objects
```

- Monitoring of memory allocation

```
> env HEAP_PROFILE_ALLOCATION_INTERVAL=1 ./mvmult_heap 20000
Starting tracking the heap
tcmalloc: large alloc 3200000000 bytes == 0x2258000 @ 0x7fd915a2eae7
0x400a71 0x400c55
Dumping heap profile to mvmult.0001.heap (3051 MB allocated cumulatively,
3051 MB currently in use)
Dumping heap profile to mvmult.0002.heap (3051 MB allocated cumulatively,
3051 MB currently in use)
Dumping heap profile to mvmult.0003.heap (3052 MB allocated cumulatively,
3052 MB currently in use)
Dumping heap profile to mvmult.0004.heap (3052 MB allocated cumulatively,
3052 MB currently in use)
Size 20000; abs. sum: 10000.000000 (expected: 10000)
```

# Instrumentation of MPI Applications

- Profile data must be written to a dedicated file for each process
- Instrumentation requires addition of the following statement following MPI\_Init (*filename* must be different for each process)

```
ProfilerStart(filename);
```

- Example code accomplishing this

```
int rank;
MPI_Comm_rank(MPICOMM_WORLD, &rank);
char filename[256];
snprintf(filename, 256, "my_app%04d.prof", rank);
ProfileStart(filename);
```

# Hardware Events with Perf

- Also referred to as `perf_events`
- Integrated with Linux kernel (`sys_event_perf_open` system call)
- Hardware event categories:
  - Cache misses and accesses issued (for L1, L2, and L3 caches) with instruction/data and load/store grouping
  - TLB related (categorized into instruction/data and load/store access types)
  - Branch statistics (branch occurrence, mispredicted branch counts)
  - Cycle (total, stalled, and idle) and instruction (issued, retired) counts
  - Node-level prefetches, loads, and stores
  - “uncore” events, collected by CPU logic shared by all cores (memory controller and NUMA related, last level cache accesses, coherency traffic, power)
- Kernel software events
  - Context switches and migrations, alignment faults, page faults, BPF events

# Perf Usage

perf <command> [<application> <arguments>]

where supported commands are:

- list: outputs events supported on local platform
- stat: profiles specified application
- record: enables per thread, per process or per CPU profiling
- report: performs analysis of recorded data
- annotate: correlates profiling data to assembly code
- top: displays statistics in real time for a running application
- bench: runs tests using predefined benchmark kernels

# Perf Example (I)

## Matrix-vector multiplication, row-major

```
> perf stat ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)

Performance counter stats for './mvmult 20000':

      1219.404556      task-clock (msec)          #      1.000 CPUs utilized
                  1      context-switches          #      0.001 K/sec
                  0      cpu-migrations          #      0.000 K/sec
      781,490      page-faults             #      0.641 M/sec
  3,898,266,727      cycles                #      3.197 GHz
  2,283,166,328      stalled-cycles-frontend   #    58.57% frontend cycles idle
  1,372,252,385      stalled-cycles-backend    #    35.20% backend  cycles idle
  3,764,331,355      instructions           #      0.97  insns per cycle
                                         #      0.61  stalled cycles per insn
  495,220,268      branches               #    406.116 M/sec
     815,338      branch-misses         #      0.16% of all branches

  1.219967824 seconds time elapsed
```

# Perf Example (II)

Matrix-vector multiplication, column-major

```
Performance counter stats for './mvmult 20000 t':  
  
 12212.530334      task-clock (msec)          # 1.000 CPUs utilized  
        11      context-switches            # 0.001 K/sec  
         0      cpu-migrations           # 0.000 K/sec  
 1,213,417       page-faults             # 0.099 M/sec  
42,933,883,759      cycles                # 3.516 GHz  
39,567,001,587      stalled-cycles-frontend   # 92.16% frontend cycles idle  
37,181,761,140      stalled-cycles-backend    # 86.60% backend  cycles idle  
 6,077,067,370      instructions           # 0.14  insns per cycle  
                           # 6.51  stalled cycles per insn  
 918,790,187       branches               # 75.233 M/sec  
 1,276,503       branch-misses          # 0.14% of all branches  
  
12.213751102 seconds time elapsed
```

# Perf Example (III)

## Matrix-vector multiplication, custom events

```
> perf stat -B -e cache-misses,dTLB-load-misses,iTLB-load-misses ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)

Performance counter stats for './mvmult 20000':

    29,307,244      cache-misses
        3,121,156      dTLB-load-misses
            4,224      iTLB-load-misses

    1.227144489 seconds time elapsed
```

## Row-major

```
Performance counter stats for './mvmult 20000 t':

    79,004,606      cache-misses
    405,044,765      dTLB-load-misses
        33,124      iTLB-load-misses

    12.185000849 seconds time elapsed
```

## Column-major

# Performance API (PAPI)

- Developed at the Innovative Computing Laboratory at University of Tennessee
- C and Fortran library and bindings
- Provides access to hardware counters, timing functions, manipulation of event sets, and system parameter queries
- Requires manual instrumentation of user code
- Bundled utilities include:
  - `papi_avail` (lists names of preset event types)
  - `papi_native_avail` (displays names of node-level and uncore events supported locally)
  - `papi_decode` (event description in csv format)
  - `papi_clockres` (retrieves resolution of timing utilities)
  - `papi_cost` (verifies latencies of various API functions)
  - `papi_event_chooser` (determines events that may be combined without conflict)
  - `papi_mem_info` (shows information on memory hierarchy)

# PAPI Example

```
...
025 #define PAPI_CALL(fn, ok_code) do { \
026     if (ok_code != fn) { \
027         fprintf(stderr, "Error: " #fn " failed, aborting\n"); \
028         exit(1); \
029     } \
030 } while (0)
031
032 #define NEV 2
033
034 int main(int argc, char **argv) {
035     int n = 1000, trans = 0;
036     if (argc > 1) n = strtol(argv[1], NULL, 10);
037     if (argc > 2) trans = (argv[2][0] == 't');
038
039     int evset = PAPI_NULL;
040     PAPI_CALL(PAPI_library_init(PAPI_VER_CURRENT), PAPI_VER_CURRENT);
041     PAPI_CALL(PAPI_create_eventset(&evset), PAPI_OK);
042     PAPI_CALL(PAPI_add_event(evset, PAPI_DP_OPS), PAPI_OK);
043     PAPI_CALL(PAPI_add_event(evset, PAPI_VEC_DP), PAPI_OK);
044     double *m, *v, *p;
045     PAPI_CALL(PAPI_start(evset), PAPI_OK);
046     init(n, &m, &v, &p, trans);
047     long long v1[NEV], v2[NEV], v3[NEV];
048     PAPI_CALL(PAPI_read(evset, v1), PAPI_OK);
049     mult(n, m, v, p, trans);
050     PAPI_CALL(PAPI_read(evset, v2), PAPI_OK);
051     double s = cblas_dasum(n, p, 1);
052     PAPI_CALL(PAPI_stop(evset, v3), PAPI_OK);
053     printf("Size %d; abs. sum: %f (expected: %d)\n", n, s, (n+1)/2);
054     printf("PAPI counts:\n");
055     printf("    init: event1: %15lld event2: %15lld\n", v1[0], v1[1]);
056     printf("    mult: event1: %15lld event2: %15lld\n", v2[0]-v1[0], v2[1]-v1[1]);
057     printf("    sum: event1: %15lld event2: %15lld\n", v3[0]-v2[0], v3[1]-v2[1]);
058     return 0;
059 }
```

# PAPI Example: Compilation and Results

## Compilation and linking with reference BLAS library

```
> gcc -O2 mvmult_papi.c -o mvmult_papi -lcblas -lpapi
```

## Results

```
> ./mvmult_papi 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
PAPI counts:
init: event1:          0 event2:          0
mult: event1:    804193640 event2:          0
sum:  event1:    20276 event2:          0
```

## Event counts after BLAS is replaced by Intel Math Kernel Library

```
PAPI counts:
init: event1:          0 event2:          0
mult: event1:    1055372246 event2:    527686123
sum:  event1:    24674 event2:    12337
```

# Tuning and Analysis Toolkit (TAU)

- Developed at the Performance Research Laboratory at University of Oregon
- Operating environments include 32- and 64-bit x86 Linux clusters, ARM platforms, Windows machines, Cray computers with CNL, IBM BlueGene and POWER running AIX or Linux, NEC SX, and GPUs (by AMD, Nvidia or Intel)
- Toolkit performs instrumentation for profiling or tracing, measurements, analysis, and visualization of performance data
- Provides a Java GUI tool (*paraprof*) for accessing collected performance profiles or databases, and for data mining
- Supports C, C++, Fortran, UPC, Python, Java, and Chapel
- Two event classes: atomic and interval
- Three instrumentation methods:
  - Source level (manual or using Program Database Toolkit)
  - Library level (linking/preloading instrumentation wrapper library)
  - Binary level (requires Dyninst tool to rewrite executable code)

# TAU Example

- Makefile selection

```
> export TAU_MAKEFILE=/opt/tau/x86_64/lib/Makefile.tau-memory-phase-papi-  
mpi-pthread-pdt
```

- Selective instrumentation configuration (select.tau)

```
BEGIN_EXCLUDE_LIST  
void cblas_dasum(int, double *, int)  
END_EXCLUDE_LIST  
  
BEGIN_FILE_EXCLUDE_LIST  
*.so  
END_FILE_EXCLUDE_LIST  
  
BEGIN_INSTRUMENT_SECTION  
loops file="mvmult.c" routine="mult"  
memory file="mvmult.c" routine="init"  
END_INSTRUMENT_SECTION
```

- PDT instrumentation

```
> taucc -tau:verbose -tau:pdtinst -optTauSelectFile=select.tau mvmult.c -O2  
-o mvmult -lcblas -lm
```

- Environment variables

```
> TAU_METRICS=TIME  
> TAU_PROFILE=1
```

# Visualization with Paraprof

Main window

Execution phases window

The screenshot shows the Paraprof visualization interface. The main window displays a hierarchical tree of applications under 'Applications' and a detailed table of trial fields. The execution phases window shows a timeline of metric values for a specific phase.

**Main window:**

- File Options Help
- Applications
  - Standard Applications
    - Default App
      - Default Exp
        - perf/maciek/home/
      - TIME

TrialField	Value
Name	perf/maciek/home/
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	4
CPU MHz	3193.000
CPU Type	Intel(R) Xeon(R) CPU X5672...
CPU Vendor	GenuineIntel
CWD	/home/maciek/perf
Cache Size	12288 KB
Command Line	/mvmult 20000
Executable	/home/maciek/perf/mvmult
File Type Index	1
File Type Name	TAU profiles
Hostname	iugis
Local Time	2017-02-13T17:20:18-05:00
Memory Size	24733500 kB
Node Name	iugis
OS Machine	x86_64
OS Name	Linux
OS Release	4.8.17-gentoo
OS Version	#1 SMP PREEMPT Wed Jan ...
Starting Timestamp	1487024418598913
TAU Architecture	default
TAU Config	-prefix=/home/maciek/packa...
TAU Makefile	/home/maciek/packages/tau...
TAU Version	2.26
TAU_BFD LOOKUP	on
TAU_CALLPATH	on
TAU_CALLPATH_DEPTH	2

**Execution phases window:**

Phase: .TAU application  
Metric: TIME  
Value: Exclusive

Std. Dev. |

Mean Max Min node 0

void init(int, double \*\*, double \*\*, double \*\*, int) C [{mvmult.c} {6,1}-{15,1}]  
Exclusive TIME: 0.586 seconds  
Inclusive TIME: 0.586 seconds  
Calls: 1.0  
SubCalls: 0.0

# VampirTrace

- Open source tool for profiling on distributed machines
- Supports MPI, OpenMP, CUDA, OpenCL, and hybrid environments
- May be invoked from third party packages (e.g., TAU, Dyninst)
- API for manual instrumentation also available
- Outputs information in Open Trace Format (OTF)
- Visualization performed by proprietary Vampir tool or open source toolkits
- Compiler wrappers are often installed on the target machine:
  - vtcc for C
  - vtcxx for C++
  - vtfort for Fortran

# VampirTrace Example: MPI Ping-Pong

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <unistd.h>
0004 #include "mpi.h"
0005
0006 int main(int argc,char **argv)
0007 {
0008     int rank,size;
0009     MPI_Init(&argc,&argv);
0010     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
0011     MPI_Comm_size(MPI_COMM_WORLD,&size);
0012
0013     if ( size != 2 ) {
0014         printf(" Only runs on 2 processes \n");
0015         MPI_Finalize(); // this example only works on two processes
0016         exit(0);
0017     }
0018
0019     int count;
0020     if ( rank == 0 ) {
0021         // initialize count on process 0
0022         count = 0;
0023     }
0024     for (int i=0;i<10;i++) {
0025         if ( rank == 0 ) {
0026             MPI_Send(&count,1,MPI_INT,1,0,MPI_COMM_WORLD); // send "count" to rank 1
0027             MPI_Recv(&count,1,MPI_INT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); // receive it back
0028             sleep(1);
0029             count++;
0030             printf(" Count %d\n",count);
0031         } else {
0032             MPI_Recv(&count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
0033             MPI_Send(&count,1,MPI_INT,0,0,MPI_COMM_WORLD);
0034         }
0035     }
0036
0037     if ( rank == 0 ) printf("\t\t\t Round trip count = %d\n",count);
0038
0039     MPI_Finalize();
0040 }
```

# VampirTrace Compilation

- Using MPI compiler wrapper

```
vtcc -vt:cc mpicc pingpong.c
```

- Direct linking with MPI library (may require the user to provide the location of MPI header files)

```
vtcc pingpong.c -lmpi
```

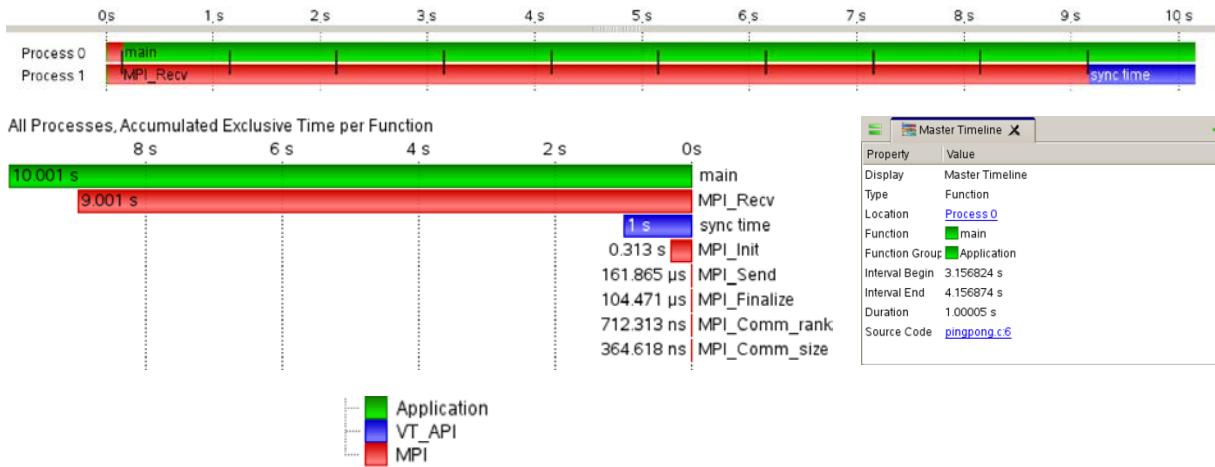
- Compiling with MPI instrumentation

```
vtcc -vt:cc mpicc -vt:mpi pingpong.c
```

- OpenMP code compilation

```
vtcc -vt:cc gcc -vt:mt -fopenmp forkjoin.c
```

# Vampir Visualization

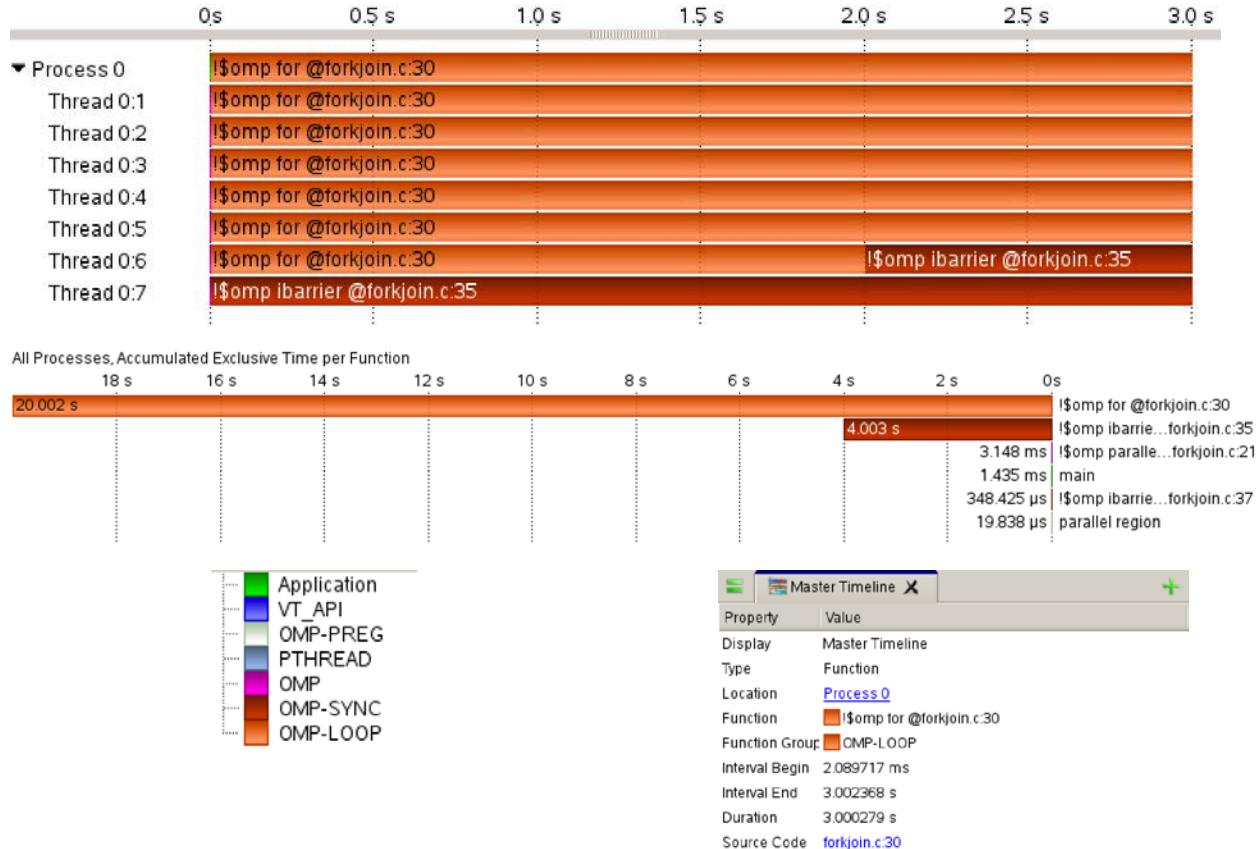


- Phase diagram shows the amount of time spent in application code (red), MPI (green), and VampirTrace API (blue)
- Black lines denote messages exchanged by the processes
- The information is displayed individually for each process (top) and cumulatively for the entire execution (bottom left)

# VampirTrace Example: OpenMP

```
0001 #include <omp.h>
0002 #include <unistd.h>
0003 #include <stdio.h>
0004 #include <stdlib.h>
0005 #include <math.h>
0006
0007 int main (int argc, char *argv[])
0008 {
0009     const int size = 20;
0010     int nthreads, threadid, i;
0011     double array1[size], array2[size], array3[size];
0012
0013     // Initialize
0014     for (i=0; i < size; i++) {
0015         array1[i] = 1.0*i;
0016         array2[i] = 2.0*i;
0017     }
0018
0019     int chunk = 3;
0020
0021 #pragma omp parallel private(threadid)
0022 {
0023     threadid = omp_get_thread_num();
0024     if (threadid == 0) {
0025         nthreads = omp_get_num_threads();
0026         printf("Number of threads = %d\n", nthreads);
0027     }
0028     printf(" My threadid %d\n",threadid);
0029
0030 #pragma omp for schedule(static,chunk)
0031 for (i=0; i<size; i++) {
0032     array3[i] = sin(array1[i] + array2[i]);
0033     printf(" Thread id: %d working on index %d\n",threadid,i);
0034     sleep(1);
0035 }
0036
0037 } // join
0038
0039 return 0;
0040 }
```

# OpenMP Vampir Visualization



- Data collected for execution on 8 threads

# Debugging

# Overview

- Tracking the origin of a parallel application execution anomaly on a supercomputer is generally much more difficult than debugging a serial application.
- Debugging an application on a high performance computer frequently requires a fairly detailed view of the supercomputer software and hardware stack to properly diagnose the anomaly.
- Several open source and commercial debugging tools and suites have been developed to assist the debugging process.
- There are several commercial parallel debuggers which support MPI and OpenMP codes.
- There are several open source serial debuggers and tool suites which can be used to debug MPI and OpenMP codes. In the case of MPI, they may require attaching several serial debuggers to a simulation.
- The GNU debugger provides multiple tools for debugging a code and enabling the user to step through the code and call stack as well as viewing variables and changing their values.

# GDB Examples: breakpoints

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc,char **argv) {
5     int i;
6     // Make the local vector size constant
7     int local_vector_size = 100;
8
9     // initialize the vectors
10    double *a, *b;
11    a = (double *) malloc(
12        local_vector_size*sizeof(double));
13    b = (double *) malloc(
14        local_vector_size*sizeof(double));
15    for (i=0;i<local_vector_size;i++) {
16        a[i] = 3.14;
17        b[i] = 6.67;
18    }
19    // compute dot product
20    double sum = 0.0;
21    for (i=0;i<local_vector_size;i++) {
22        sum += a[i]*b[i];
23    }
24    printf("The dot product is %g\n",sum);
25
26    free(a);
27    free(b);
28    return 0;
29 }
```

Breakpoint Command Type	gdb breakpoint command	Description
Break by function	break printf	Pauses the execution at line 24
Break by line number	break 17	Pauses the execution at line 17
Break by line number and filename	break dotprod_serial.c:17	Pauses the execution at line 17
Break by conditional	break dotprod_serial.c:16 if i==4	Pauses the execution at line 16 when i equals 4

```
(gdb) info breakpoints
Num  Type      Disp Enb Address          What
1   breakpoint  keep y  0x0000000000400450 <printf@plt>
2   breakpoint  keep y  0x00000000004005ef in main at dotprod_serial.c:17
3   breakpoint  keep y  0x00000000004005ef in main at dotprod_serial.c:17
4   breakpoint  keep y  0x00000000004005cf in main at dotprod_serial.c:16
stop only if i==4
(gdb) ■
```

# GDB Examples: enable, disable, delete, break

```
[(gdb) disable 2
[(gdb) enable once 3
[(gdb) delete 1
[(gdb) info breakpoints
Num      Type      Disp Enb Address          What
2        breakpoint    keep n  0x00000000004005ef in main at dotprod_serial.c:17
3        breakpoint    dis  y   0x00000000004005ef in main at dotprod_serial.c:17
4        breakpoint    keep y   0x00000000004005cf in main at dotprod_serial.c:16
stop only if i==4
[(gdb) tbreak dotprod_serial.c:24
Temporary breakpoint 5 at 0x40067b: file dotprod_serial.c, line 24.
[(gdb) info breakpoints
Num      Type      Disp Enb Address          What
2        breakpoint    keep n  0x00000000004005ef in main at dotprod_serial.c:17
3        breakpoint    dis  y   0x00000000004005ef in main at dotprod_serial.c:17
4        breakpoint    keep y   0x00000000004005cf in main at dotprod_serial.c:16
stop only if i==4
5        breakpoint    del  y   0x000000000040067b in main at dotprod_serial.c:24
(gdb) █
```

```
[(gdb) tbreak 17
Temporary breakpoint 1 at 0x4005ef: file dotprod_serial.c, line 17.
[(gdb) run
Starting program: /home/andersmw/learn/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdfc8) at dotprod_serial.c:17
17          b[i] = 6.67;
[(gdb) print i
$1 = 0
[(gdb) print a[i]
$2 = 3.1400000000000001
[(gdb) print b[i]
$3 = 0
(gdb) █
```

# GDB Examples: Watchpoints, catchpoints

```
|(gdb) b 20
Breakpoint 1 at 0x40061b: file dotprod_serial.c, line 20.
|(gdb) r
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:20
20      double sum = 0.0;
|(gdb) watch sum
Hardware watchpoint 2: sum
|(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 6.9533558074263132e-310
New value = 0
main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:21
21      for (i=0;i<local_vector_size;i++) {
|(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 0
New value = 20.9438
main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:21
21      for (i=0;i<local_vector_size;i++) {
|(gdb)
```

```
(gdb) info watchpoints
Num      Type            Disp Enb Address          What
2        hw watchpoint  keep y
                  breakpoint already hit 2 times
(gdb)
```

# GDB Examples: Backtrace

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void initialize(double *a, double *b,int local_vector_size)
5 {
6     int i;
7     for (i=0;i<local_vector_size;i++) {
8         a[i] = 3.14;
9         b[i] = 6.67;
10    }
11 }
12
13 int main(int argc,char **argv) {
14     int i;
15     // Make the local vector size constant
16     int local_vector_size = 100;
17
18     // initialize the vectors
19     double *a, *b;
20     a = (double *) malloc(
21             local_vector_size*sizeof(double));
22     b = (double *) malloc(
23             local_vector_size*sizeof(double));
24
25     initialize(a,b,local_vector_size);
26
27     // compute dot product
28     double sum = 0.0;
29     for (i=0;i<local_vector_size;i++) {
30         sum += a[i]*b[i];
31     }
32     printf("The dot product is %g\n",sum);
33
34     free(a);
35     free(b);
36     return 0;
37 }
```

```
((gdb) break 8
Breakpoint 1 at 0x40059e: file dotprod_serial.c, line 8.
((gdb) run
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, initialize (a=0x601010, b=0x601340, local_vector_size=100)
at dotprod_serial.c:8
8          a[i] = 3.14;
((gdb) backtrace
#0  0x000000000040059e in initialize (a=0x601010, b=0x601340, local_vector_size=100)
at dotprod_serial.c:8
#1  0x0000000000400643 in main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:25
(gdb) 
```

# GDB Cheat Sheet

Command	Abbreviation	Function
<b>run</b>	r	Begins execution in the debugger
<b>continue</b>	c	Continues execution in the debugger after a pause
<b>quit</b>	q	Quits the debugger
<b>break</b>	b	Sets a breakpoint
<b>watch</b>		Sets a watchpoint
<b>backtrace</b>	bt	Prints the call stack
<b>set variable</b>	set var	Sets a variable value
<b>thread</b>	t	Switches to a different thread identifier
<b>list</b>	l	Lists source code near the present stopping point

# Commercial Debuggers

Commerical Debugger	Notable Capabilities
TotalView (4)	Support for OpenMP, MPI, OpenACC, CUDA
Allinea DDT (2)	Support for OpenMP, Pthreads, MPI, CUDA
Intel Parallel Debugger (5)	Support for multicore debugging

# Tools in the Valgrind suite

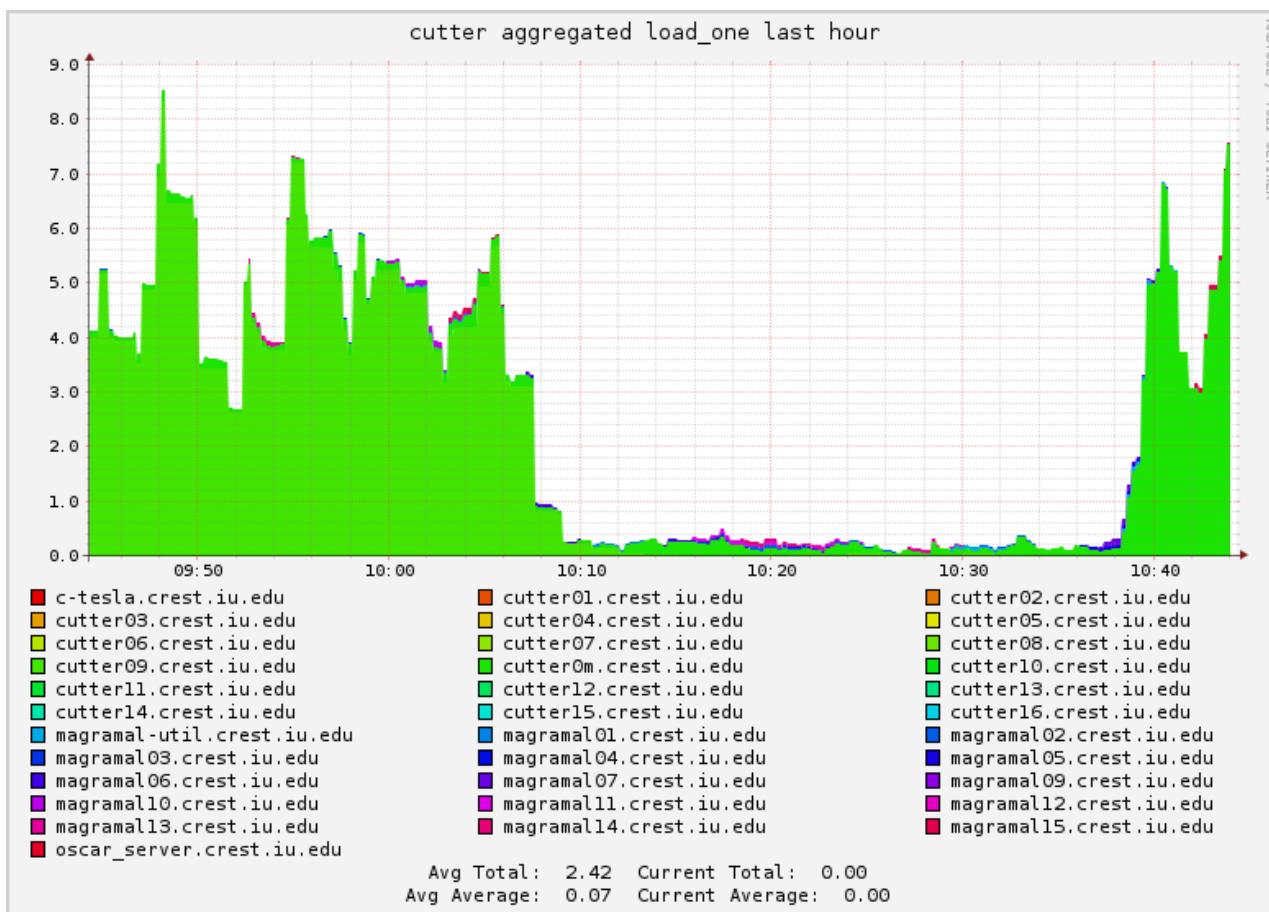
Tool	Description
Memcheck	Reports memory errors, including memory leaks or access to memory that is not yet allocated.
Cachegrind	Identifies the number of cache misses.
Callgrind	Extends cachegrind with some additional information.
Massif	Heap profiler.
Helgrind	Debugger for finding data race conditions.
DRD	Multithread debugging for C and C++ programs.

# Compiler Flags for Debugging

Action	gcc	icc	clang	pgcc
Enable pointer bounds checking (R)	-fcheck-pointer-bounds	-check-pointers-mpx=rw		-Mbounds
Enable address sanitizer (R)	-fsanitize=address		-fsanitize=address	
Enable thread sanitizer (R)	-fsanitize=thread		-fsanitize=thread	
Enable leak sanitizer (R)	-fsanitize=leak		-fsanitize=leak	
Enable undefined behavior sanitizer (R)	-fsanitize=undefined		-fsanitize=undefined	
Enable all common warning types (S)	-Wall	-Wall	-Wall	-Minform=warn
Warn if the code does not strictly comply with ANSI C or ISO C++ (S)	-pedantic		-pedantic	-Xa
Warn on use of uninitialized variables (S)	-Wuninitialized	-Wuninitialized	-Wuninitialized	
Warn when local variable shadows another variable (S)	-Wshadow	-Wshadow	-Wshadow	
Warn if comparison between signed and unsigned integer may produce wrong result (S)	-Wsign-compare	-Wsign-compare	-Wsign-compare	
Warn if undefined identifier is used in preprocessor directive (S)	-Wundef		-Wundef	
Warn when undeclared function is used or declaration doesn't specify a type (S)	-Wimplicit	-Wmissing-declarations -Wmissing-prototypes	-Wimplicit	

# System Monitors to Aid Debugging

- Ganglia



# Accelerator Architecture

# Rationale for Accelerator Devices

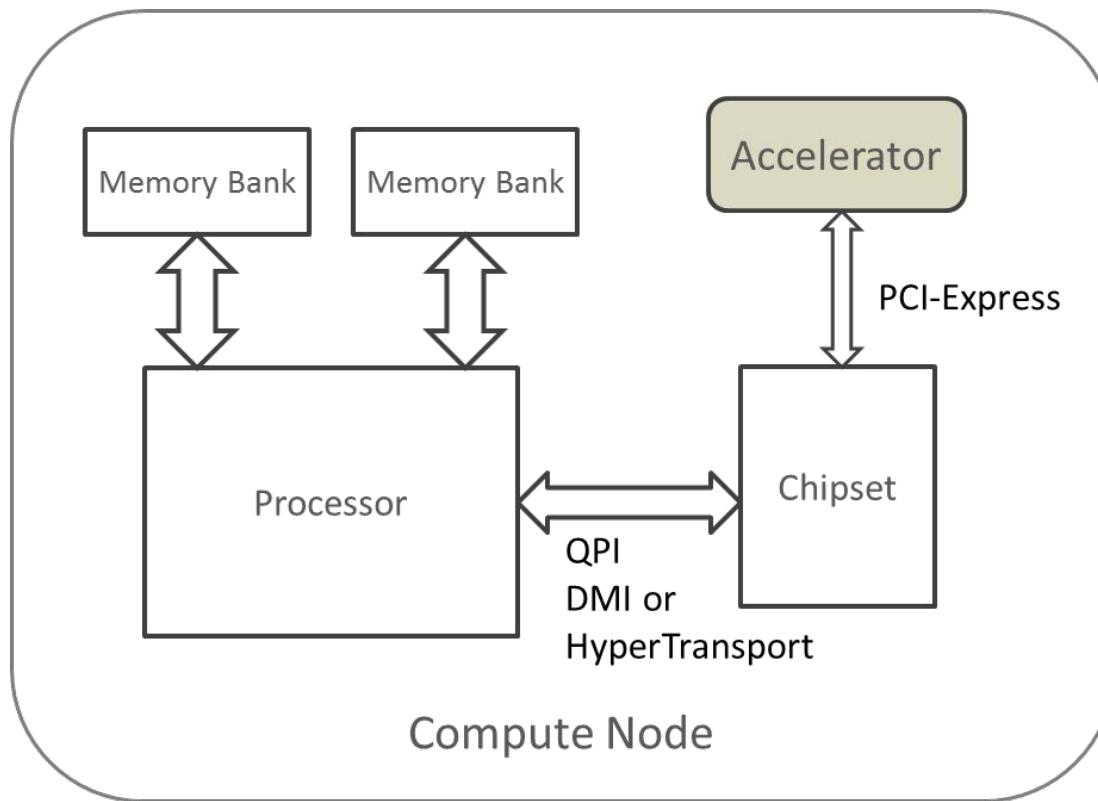
- CPUs are not optimized for specific applications
  - Generic instruction sets
  - Support diverse workload types
  - Implementation a compromise between supported features, physical constraints, and final product price
  - Limited transistor budget per chip
  - Power constraints
- Custom architectures potentially provide multiple benefits
  - Designed and optimized for specific function or task
  - Drastically improved performance for “native” applications
  - Better power characteristics
  - Improved performance per Watt
  - Better I/O integration (pin counts, connector types, protocol support, bandwidth matching of attached peripherals, etc.)
- May be manufactured relatively cheaply if deployed in volume
- Easy integration when leveraging industry standard interfaces

# Accelerator Drawbacks

- Not generic purpose
  - Poor performance for workload classes inconsistent with device's purpose
- Heterogeneous
  - Different architecture and execution model to that of host processor
  - Impose refactoring of application code into accelerated and non-accelerated (i.e., executed on conventional CPU) segments
  - Require specialized kernel drivers
- May be more difficult to program and optimize
  - May require custom language extensions and libraries
  - Knowledge of underlying architecture details is usually necessary to write a good code
  - Imposes explicit management of computations on the programmer (data and work offload, device identification and setup)
  - Limited portability to other device classes or even between accelerators of the same type but from different vendors
  - Potentially steep learning curve for the uninitiated
- Additional system component
  - Hardware cost
  - Another energy sink (make sure your power supply is sufficient)
  - Integration issues (space, cooling, interface availability)

# Placement in a Conventional System

- Use industry-standard interfaces (typically PCIe)
- Modern CPUs provide PCIe endpoints directly on die



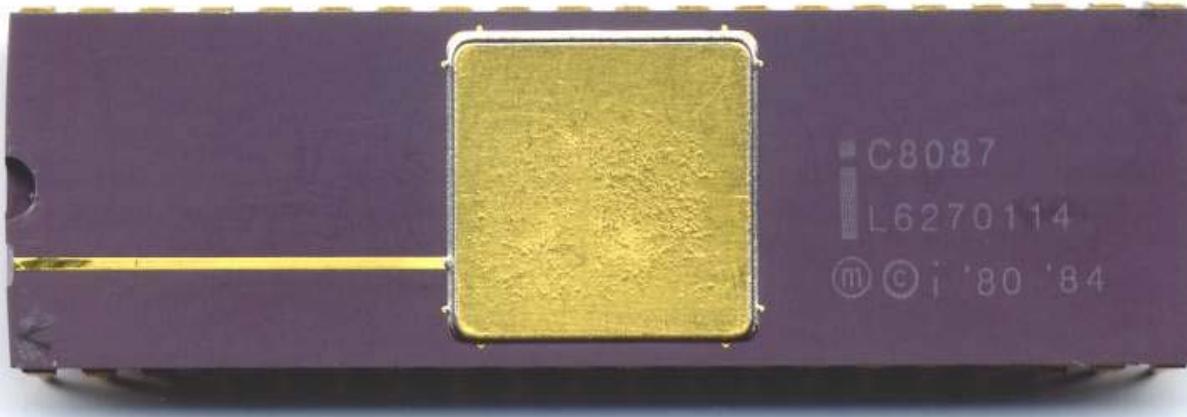
# Common Accelerator Types

- Graphics and video processing
- Arithmetic
  - Double-precision floating-point
- Multimedia processing
  - Encoding
  - Decoding
  - Multiplexing
- Digital Signal Processing (DSP)
- Cryptography
- I/O, smart DMA
- Artificial Intelligence

# Early Accelerators: Coprocessors

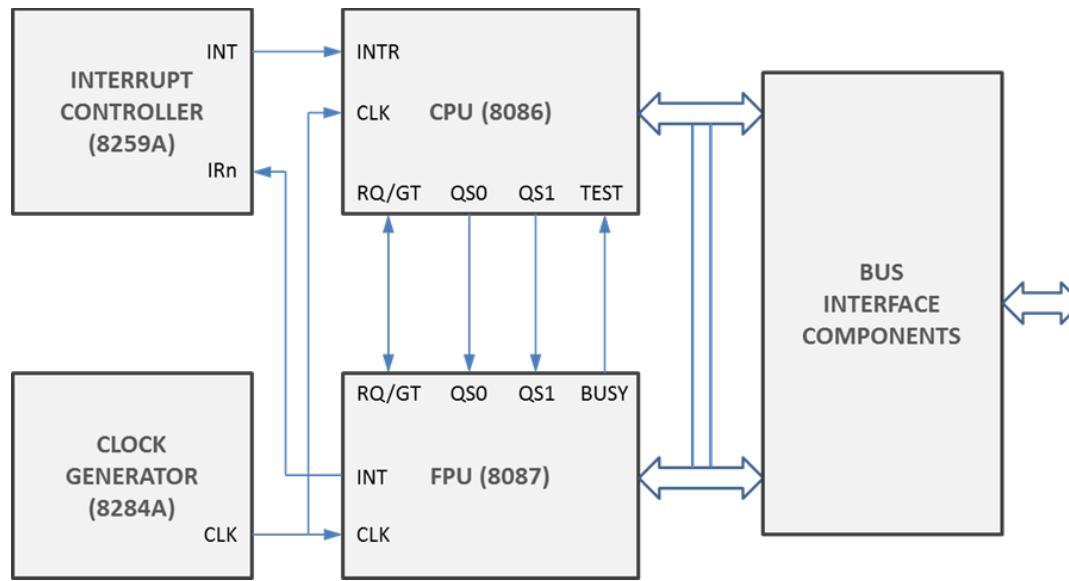
- Date back to 1970s
- Initially a separate device
  - Required a dedicated socket or pad pattern to solder a chip
  - Specialized connecting interface (data and control)
  - Custom access sequence
- Later, included on the same die as the main processor (e.g., System on Chip) when logic density sufficiently increased
- Tightly integrated and cooperating with the host processor
  - Hardware controlled offload by
    - Special instruction format and arrangement of control signals or
    - Memory mapped access to control and data registers
  - Custom synchronization mechanism
- Very limited portability between processor families

# Example: Intel 8087



- Released by Intel in 1980
- Numerical floating-point coprocessor for the i8086 family
  - Single- (32-bit), double- (64-bit), and extended-precision (80-bit) floats
  - 16, 32, and 64-bit signed binary integers, and 80-bit BCD integers
  - About 60 instructions (add/subtract, multiply/divide, sq. root, trigonometric functions, base 2 logarithm, conversions)
- Influenced by the upcoming IEEE754 standard, but not compliant
- Implemented in 3µm HMOS process, 45,000 transistors
- Achieved 50kflops using 2.4 Watts (speed variants from 4 to 10MHz)
- Cloned by AMD and Cyrix

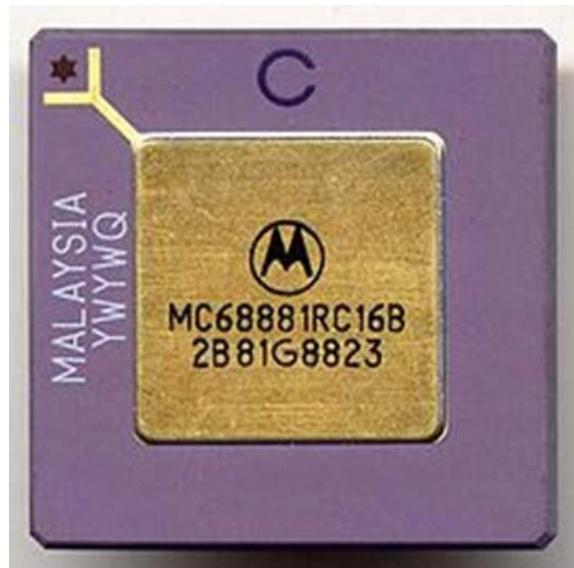
# Intel 8087 System Interface



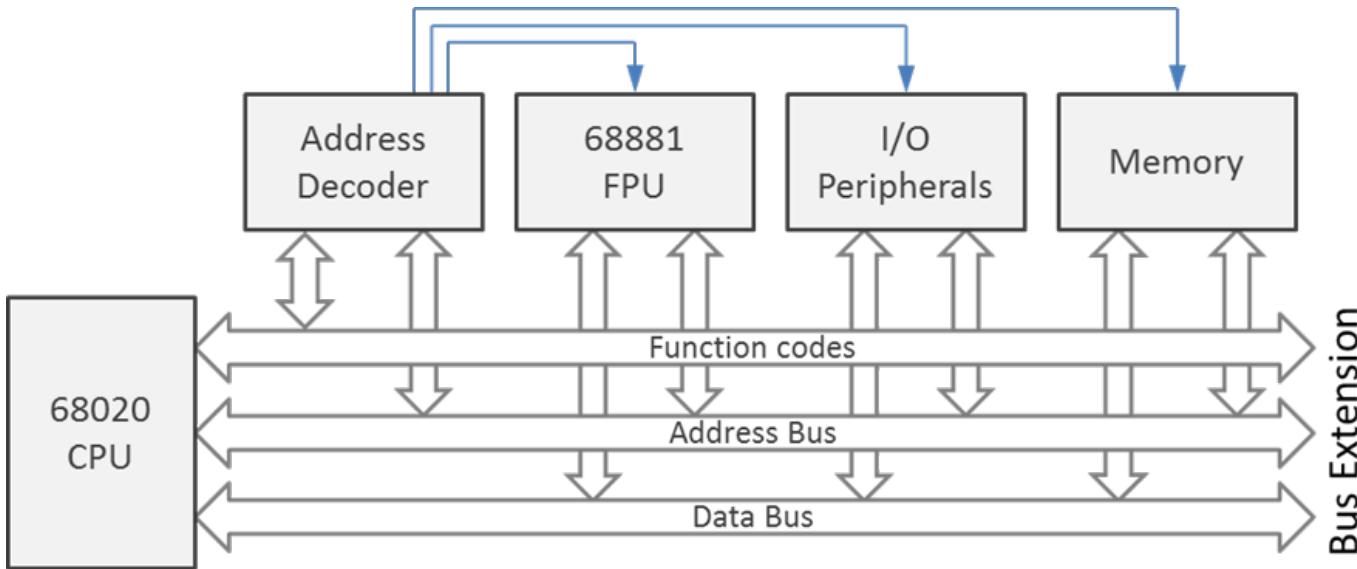
- Shares address and data buses with the main CPU
- “Escape” instructions redirect work to coprocessor
- Memory operands required the main CPU to perform a dummy memory access
- Concurrent operation with CPU crash prone due to potential bus conflicts
- In practice, the CPU has to wait for the coprocessor to finish the computation (FWAIT instruction)

# Example: Motorola MC68881

- Introduced by Motorola in 1984
- Companion to MC68000 family with 32-bit data buses
- Implemented in an HCMOS process using 155,000 transistors
- Floating-point coprocessor
  - Single, double, and extended precision floats
  - 96-bit packed BCD floats
  - 8, 16, 32-bit integers
  - IEEE754 compliant
  - Add/subtract, multiply/divide, sq. root, trigonometric and inverse functions, hyperbolic functions, exponentiation, logarithms
- Achieved 240kfps at 25MHz consuming 0.75 Watts



# MC68881 System Interface



- Memory-mapped control registers
- Uses standard system bus
- Coprocessor may use all addressing modes supported by the main CPU
- Coprocessor may run at a different clock speed than CPU
- Single CPU may control up to eight coprocessors

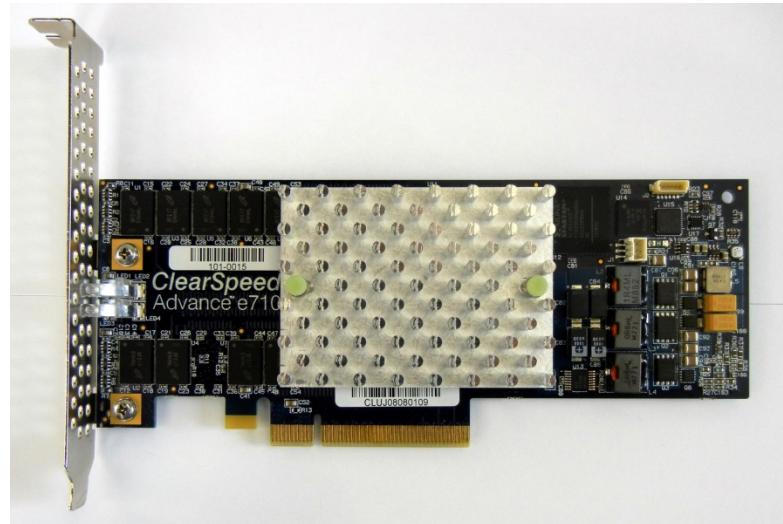
# Example: Weitek 3167

- Memory-mapped interface
  - Communication via memory move instructions
- Often used interposition socket due to incompatible pin arrangement with existing coprocessors
- The fastest FPU on the market at release time
  - Linpack: 1.36 (SP) and 0.6 (DP) Mflops
  - Whetstone: 5.6 (SP) and 3.7 (DP) Mflops
  - Dissipated 1.84W at 25MHz
- Replaced by W4167 with approximately 3x better performance (shown)



# Example: Clearspeed Advance

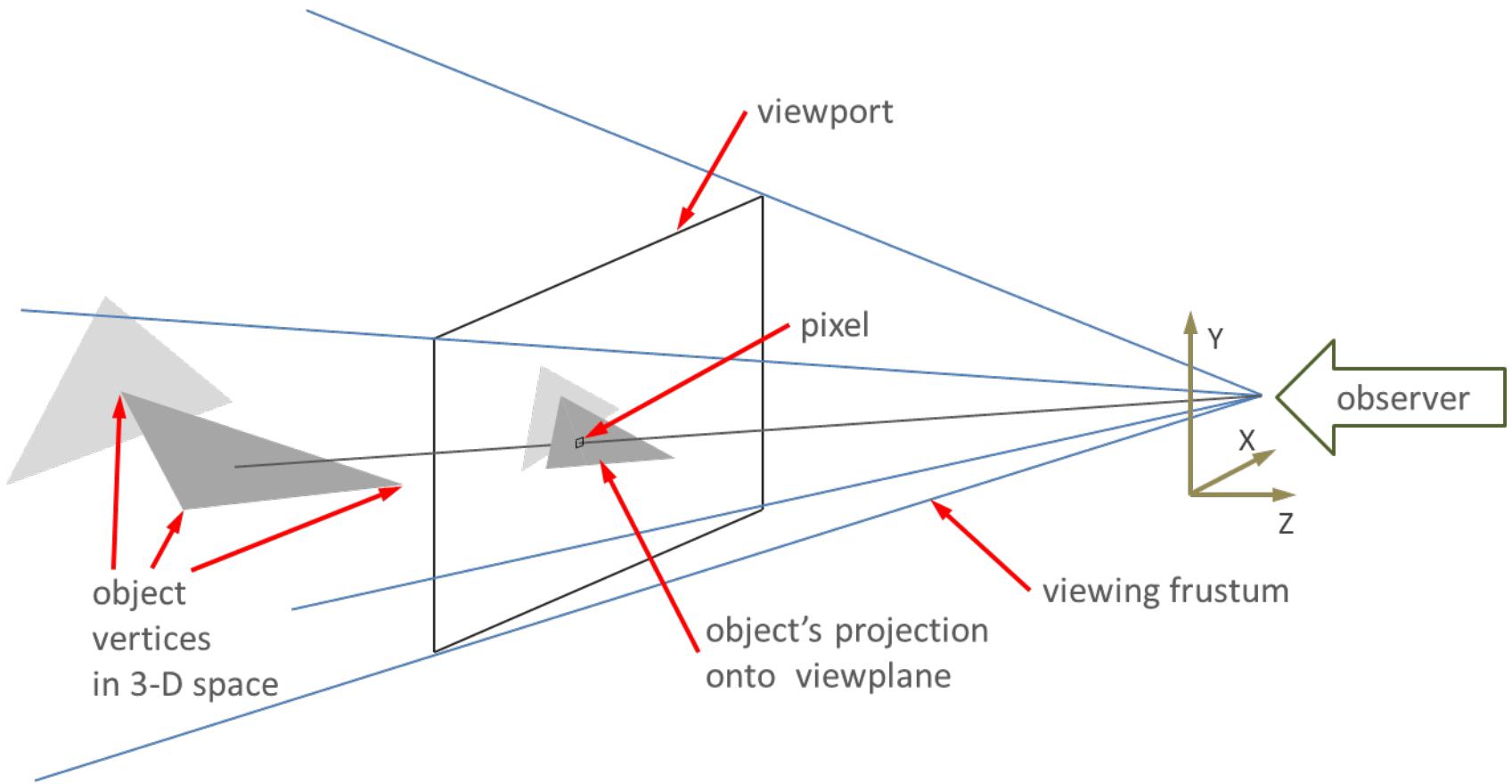
- Use industry-standard interfaces
- Version X620 equipped with PCI-X interface (2005)
  - Dual CSX600 SIMD processors, each comprising 96 double-precision processing elements
  - 128 million transistors in 130nm process
  - Sustained 33Gflops at 250MHz while dissipating 10W
  - 1GB ECC on-board memory
  - Employed in Tsubame cluster (7<sup>th</sup> place on Top500 in 2006)
- PCIe connectivity used in version e710 (2008 – shown)
  - Dual CSX700 processors
  - 96Gflops at 12W
  - 2GB memory



# Introduction to Graphics Processing Units

- Optimized for generation of 2D perspective projections of 3D objects
- Need to correctly orient and scale objects, figure out occlusion by other objects, and compute object color attenuation due to shading, incident light color and intensity, and object surface properties (reflectivity, etc.)
- Types of computation include floating-point matrix-vector multiplication, dot products, weighted vector sum
- Multiple computations may be required for each pixel in the frame due to antialiasing
- Additional support is needed for textures, environment-enhanced mapping, fog, and other effects
- Smooth video rendering therefore requires a large number of floating-point operations per second
- Some of the realism may be sacrificed to obtain the desired rendering speed

# Elements of a 3D Scene



# GPU Protoplasts

- Player-missile or sprite graphics in 8-bit computers
- 2D graphics adapters popularized by IBM PC
  - Color Graphics Adapter popularized by IBM PC (two colors at 640x200 pixels or 80x25 text mode)
  - Enhanced Graphics Adapter (16 simultaneous colors, 640x350 pixels)
  - Video Graphics Array (640x400/16 colors or 320x200/256 colors, 18-bit RGB values)
- Self-contained solutions:
  - TMS34010 by Texas Instruments (fixed-function 2D accelerator, drawing primitives in software)
  - IBM 8514 (hardware line drawing, area fills, block transfers, raster operations)

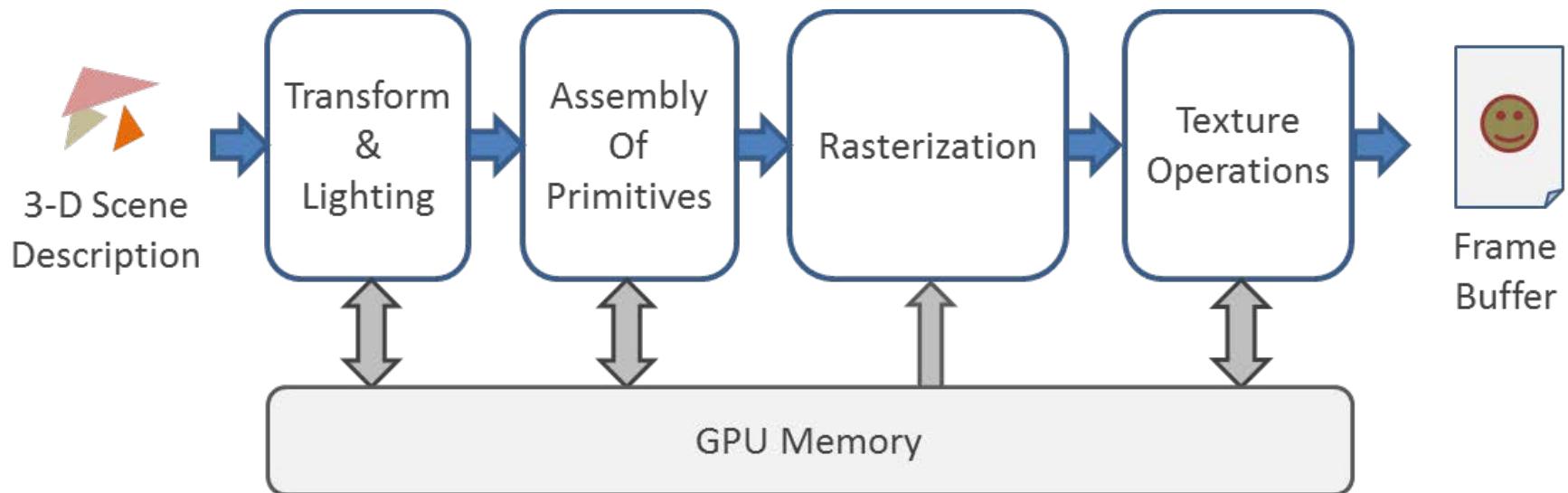
# Early GPUs

- Silicon Graphics Inc. workstations
  - 3D rendering inspired by Stanford Geometry Engine
  - 100s of thousands (1991) to >10 million polygons per second
- Add-on 3D rendering cards
  - Voodoo line from 3dfx Interactive
- Combined 2D and 3D acceleration
  - ViRGE by S3, Rage from ATI, etc.

# Growth of GPU Capabilities

- Standard 2D hardware acceleration
  - Line drawing, polygon fills, BitBLTs
- 3D rendering features
  - Perspective transformation
  - Shading: flat and Gouraud
  - Texture mapping and filtering
  - MIP-mapping
  - Alpha-blending (translucency)
  - Depth queuing
  - Fogging
  - Z-buffering
  - Simple video decoding support (e.g., MPEG-1)
  - T&L: transform, clipping, and lighting support in hardware
  - Stencil buffers (shadow and reflection rendering)
  - Anti-aliasing
  - High dynamic range (HDR)
  - Advanced video codecs

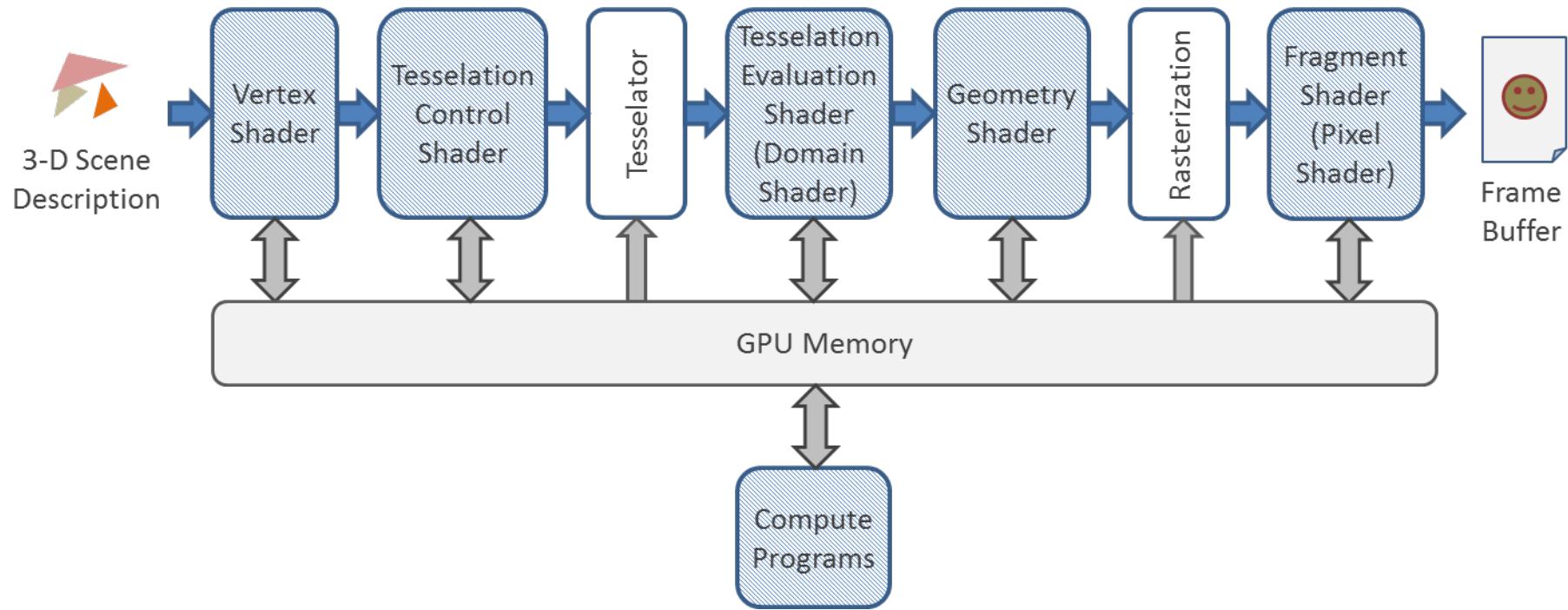
# Fixed-Function Graphics Pipeline



# Modern GPU Attributes

- Unified shaders
  - Introduced by Nvidia's GeForce 8 and AMD's Radeon HD2000 series
  - Fully programmable by user/video driver developer
  - No longer assigned to perform a predefined function at a specific pipeline stage
- Shared internal storage accessible to shader code further improves processing efficiency (scratchpads, caches)
- Double-precision arithmetic added in 2007 enables broad class of scientific computations (but not always in strict compliance with IEEE754)
- Microarchitecure optimizations for
  - Tessellation, primitive discard acceleration
  - Asynchronous processing
  - Independent scheduling and work dispatch
  - Offload management
  - Fine-granularity preemption
- 2D operations now emulated by 3D processing hardware
- Availability of higher-priced and more stable GPGPUs targeting computational workloads in addition to consumer line of products

# Modern Rendering Pipeline



# Comparison of Recent GPUs

Company	Product Name	Product Type	Transistor Count [billion]	Base Clock [MHz]	Peak FP GFlops		Memory Bandwidth [GB/s]	Memory Bus Width	Max. Power [W]
					SP	DP			
AMD	Radeon R9 Fury X	GPU	8.9	1050	8601.6	537.6	512	4096	275
AMD	FirePro W9100	Accel.	6.2	930	5240	2620	320	512	275
Nvidia	GeForce GTX Titan X	GPU	8	1000	6144	192	336	384	250
Nvidia	K40	Accel.	7.1	745	5040	1680	288	384	235

# Modern GPU Example: Nvidia Pascal

- Released in 2016
- Increased number of double-precision FPUs
- Addition of “half-precision” floating-point mode for deep learning, sensor processing, and radio astronomy applications
- Second generation, stacked high-bandwidth memory (HBM2) for increased capacity and bandwidth
- Simplified (to the user) offload mechanism through unified access to system memory and demand paging
- High-speed NVlink bus
- Improved performance of atomic operations targeting data in on-chip memories and external DRAM

# Pascal GPU Parameters

Parameter	Value
Technology node	TSMC 16 nm Fin-FET
Transistor count	15.3 billion
Die size	610 mm <sup>2</sup>
Clock frequency	1328 MHz (1480 MHz boost)
Memory type	HBM2
Memory size	16 GB
Memory bus width	8x512 bits
Memory bandwidth	720 GB/s
Thermal design power	300 W
Resource counts (per GPU)	
Streaming multiprocessors (SMs)	56 (max. 60)
CUDA cores	3584
Texture Processing Clusters (TPCs)	28
Texture units	224
Register file size	14336 KB
L2 cache size	4096 KB
FP32 CUDA cores	2584
FP64 CUDA cores	1792
Aggregate performance	
Peak double-precision	5.3 TFlops
Peak single-precision	10.6 TFlops
Peak half-precision	21.2 TFlops

# Pascal Architecture (I)

- Pascal chip comprises
  - Stream Multiprocessors (SMs) grouped in clusters (GPCs)
  - Eight 512-bit wide HBM2 interfaces associated with memory controllers
  - Global thread scheduling engine
  - L2 Caches
  - 16-lane PCIe 3.0 host interface
  - 4 NVlink interfaces
- Each SM includes
  - 64 single- and 32 double-precision CUDA cores
  - Instruction buffer
  - Thread scheduler
  - Two dispatch units
  - 128KB register file
  - 64KB L1 data cache

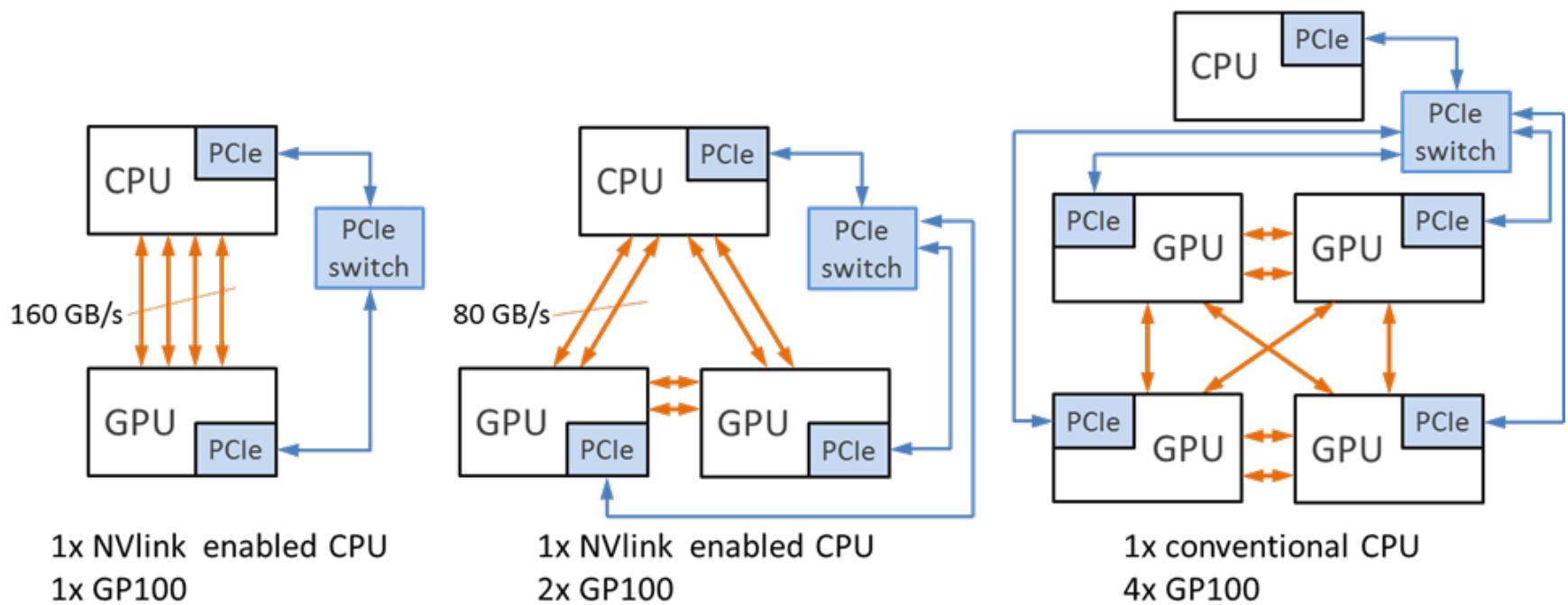
# Pascal Architecture (II)



# Details of Stream Multiprocessor



# Sample NVlink Topologies



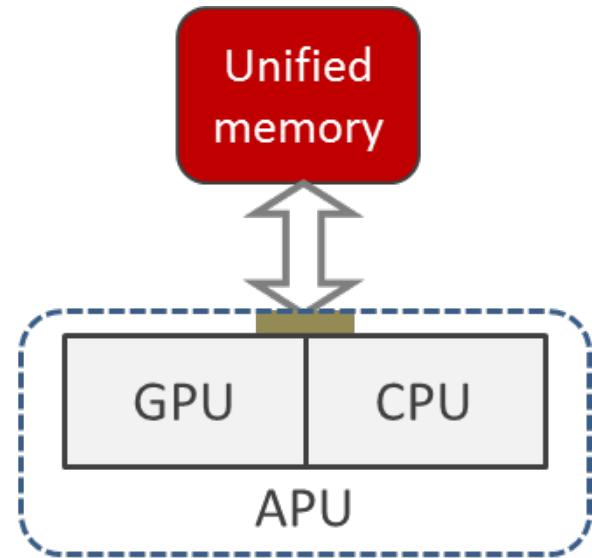
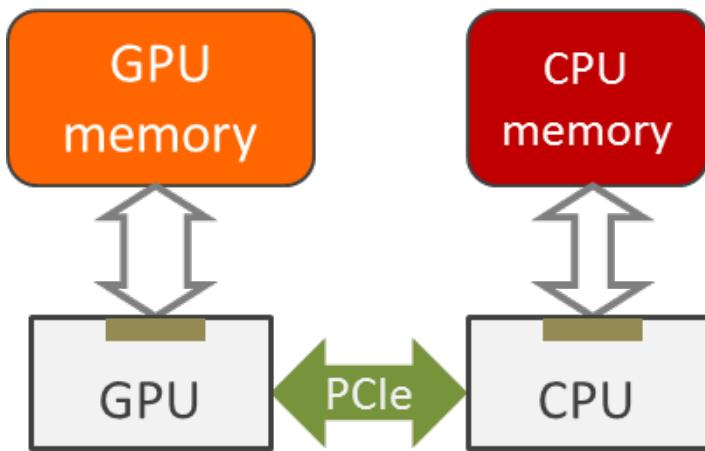
# GPU Programming Environments

- CUDA
  - Modified C++ compiler
  - API and library
  - GPU-specific data attributes and task syntax
  - Limited to Nvidia GPUs
- OpenACC
  - Directive based (#pragma)
  - Custom compiler and API
  - Focus on simplicity and portability
- OpenCL
  - C-like programming language and API
  - Supports program development on all elements of heterogeneous platform and defines interactions between them
  - Higher level data types and qualifiers for events, functions, address spaces, and access
  - Operator overloads for vector operations
  - Extensive library of mathematic, geometric, vector, image, memory, and synchronization functions

# Heterogeneous System Architecture

- Developed by non-profit HSA Foundation including industry and academic members
- Provides ISA-independent runtime and system architecture APIs
- Identifies two types of compute units
  - Latency Compute Unit (conventional CPU)
  - Throughput Compute Unit (accelerator)
- Both unit types share cache-coherent physical memory implementing a unified virtual address space
  - Shared page tables
  - Data access accomplished through pointer passing (no data copies!)
  - Page fault support
  - Limited reliance on system calls
  - Improved task queueing at execution devices
- Found in Systems on Chips (SoCs) and AMD Accelerated Processing Units (APUs)

# Comparison of System Topologies with Discrete and Unified Accelerators



# AMD APU Parameters

Architecture Codename	Fabrication Process [nm]	Die Size [mm <sup>2</sup> ]	CPU			GPU		Memory Support	Max. TDP [W]
			Architecture	Clock [GHz]	Max. Cores	Clock [MHz]	Shaders		
Kaveri	28 nm	245	Steamroller	4.1/4.3	4	866	512	DDR3-2133	95
Carrizo	28 nm	245	Excavator	2.1/3.4	4	800	512	DDR3-2133	35
Bristol Ridge	28 nm	250	Excavator	3.7/4.2	4		512	DDR4-2400	65

# The Essential OpenACC

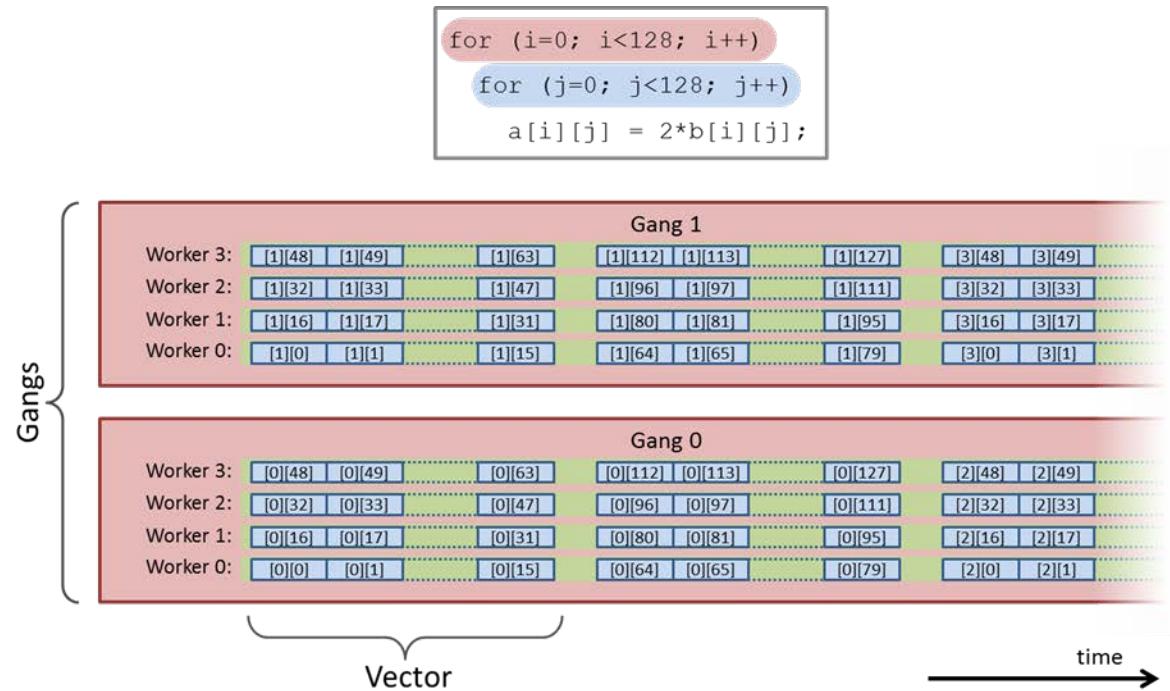
# Common GPU Programming Environments

- CUDA (Compute Unified Device Architecture)
  - C, C++, and Fortan language support (custom compilers)
  - Limited to Nvidia devices (GeForce, Quadro, Tesla)
  - Optimized libraries for FFT, BLAS, dense and sparse solvers, graph analytics, random number generation, and physics simulation
- OpenCL
  - Heterogeneous environment support (including host and devices)
  - ISO C99 and C++14 programming API
  - Distinguishes four levels of memory hierarchy on accelerators
- C++ AMP
  - Set of extensions and C++ compiler developed by Microsoft
  - Specifies two devices types for function execution, *cpu* and *amp*
  - N-dimensional array data and index objects with associated *views*
- OpenACC
  - Directive based, discussed further here

# Open Accelerators (OpenACC)

- “Directives for accelerators” (pragmas)
- Directives are ignored if compiler cannot support them
- Initial specification created by PGI, CAPS entreprise, Cray, and Nvidia in 2011
- C, C++, and Fortran languages
- Focus on simplicity and portability
- User explicitly identifies sections of code that may be offloaded to accelerator
- Most recent API specification is v2.5 (Oct. 2015)
- Available compilers include PGI, Pathscale, GCC 6, OpenUH, and OpenARC

# Parallel Workload Decomposition



- Gangs
- Workers
- Vectors

# Parallel Execution Semantics

- Compute region execution on accelerator starts in gang-redundant mode (GR)
  - Each gang executes the same code
- Once a parallel code is encountered, the execution switches to gang-partitioned (GP) mode
  - Loop iterations are distributed across gangs
  - Only one worker active per gang (worker-single, or WS mode)
  - If only one vector lane is used by the worker, it proceeds in vector-single (VS) mode
- If parallel region (or its section) is marked for worker-level work sharing, it switches to worker-partitioned mode (WP)
  - GP and WP modes may be activated at the same time
- Frequently processing may benefit from vector-level parallelization (e.g., leveraging SIMD units), thus initiating vector-partitioned (VP) mode
  - VP mode may be activated concurrently with any combination of gang and worker modes

# Common OpenACC Library Calls

- *acc\_get\_num\_devices* obtains the number of attached accelerators
- *acc\_get\_device\_type* indicates current device type
- *acc\_set\_device\_type* sets the preferred device type for execution of parallel regions of code
- *acc\_get\_device\_num* returns the index of device of specified type
- *acc\_set\_device\_num* selects specific device to execute parallel code
- Device types typically include: *acc\_device\_nvidia*, *acc\_device\_radeon*, and *acc\_device\_xeonphi*
- Macro \_OPENACC may be used to check the implementation version

# OpenACC Library Calls Example

- Code

```
001 #include <stdio.h>
002 #include <openacc.h>
003
004 int main() {
005     printf("Supported OpenACC revision: %d.\n", _OPENACC);
006
007     int count = acc_get_num_devices(acc_device_nvidia);
008     printf("Found %d Nvidia GPUs.\n", count);
009     int n = acc_get_device_num(acc_device_nvidia);
010     printf("Default accelerator number is %d.\n", n);
011
012     return 0;
013 }
```

- Output (executed on Cray XK7)

```
Supported OpenACC revision: 201306.
Found 1 Nvidia GPU(s).
Default accelerator number is 0.
```

# OpenACC Environment Variables

- `ACC_DEVICE_TYPE` sets the default device type for parallel execution
- `ACC_DEVICE_NUM` determines the index of default device to be used
- `ACC_PROFILIB` selects path to the profiling library

# The parallel Directive

```
#pragma acc parallel [clause-list]  
structured-block
```

- Identifies *structured-block* as parallel execution region
- It creates one or more gangs
- The number of gangs, workers, and vector is constant within the region
- The execution is terminated by an implicit barrier at the end of the region (all parallel work must finish before the following code statements are executed)

# Select Clauses of parallel Directive

- **async** [*(integer-expression)*] permits the host to immediately execute the code following the parallel region (removes the barrier)
- **wait** [*(integer-expression-list)*] blocks the current host thread until all listed asynchronous regions complete their execution
- **num\_gangs**(*integer-expression*) specifies the number of gangs to be used
- **num\_workers**(*integer-expression*) requests specific number of workers per gang
- **vector\_length**(*integer-expression*) assigns specific number of vector lanes to each worker

# Asynchronous Execution Example

- Code:

```
001 #include <stdio.h>
002
003 const int N = 1000;
004
005 int main() {
006     int vec[N];
007     int cpu_sum = 0, gpu_sum = 0;
008
009     // initialization
010     for (int i = 0; i < N; i++) vec[i] = i+1;
011
012     #pragma acc parallel async
013     for (int i = 100; i < N; i++) gpu_sum += vec[i];
014
015     // the following code executes without waiting for GPU result
016     for (int i = 0; i < 100; i++) cpu_sum += vec[i];
017
018     // synchronize and verify results
019     #pragma acc wait
020     printf("Result: %d (expected: %d)\n", gpu_sum+cpu_sum, (N+1)*N/2);
021
022     return 0;
023 }
```

- Output:

```
Result: 500500 (expected: 500500)
```

# The kernels Directive

```
#pragma acc kernels [clause-list]
structured-block
```

- Converts *structured-block* into sequence of parallel kernels
- The number of gangs, workers, and vector lanes may be different for each kernel
- Typically one kernel is created for each loop nest
- The parallelization parameters are determined by the compiler (may be more convenient for beginners)

# Example of kernels Directive

```
001 #include <stdio.h>
002 #include <stdlib.h>
003
004 const int N = 500;
005
006 int main() {
007     // initialize triangular matrix
008     double m[N][N];
009     for (int i = 0; i < N; i++)
010         for (int j = 0; j < N; j++)
011             m[i][j] = (i > j)? 0: 1.0;
012
013     // initialize input vector to all ones
014     double v[N];
015     for (int i = 0; i < N; i++) v[i] = 1.0;
016
017     // initialize result vector
018     double b[N];
019     for (int i = 0; i < N; i++) b[i] = 0;
020
021     // multiply in parallel
022     #pragma acc kernels
023     for (int i = 0; i < N; i++)
024         for (int j = 0; j < N; j++)
025             b[i] += m[i][j]*v[j];
026
027     // verify result
028     double r = 0;
029     for (int i = 0; i < N; i++) r += b[i];
030     printf("Result: %f (expected %f)\n", r, (N+1)*N/2.0);
031
032     return 0;
033 }
```

# Example of kernels Directive (cont.)

- Fixed size matrix-vector multiplication
- Two level loop nest executed on accelerator (lines 23-25)
- Output:

```
Result: 125250.000000 (expected 125250.000000)
```

# Data Offload Management

- **copy(*variable-list*)**
  - Performs data copies on entry and exit from parallel region
  - On region entry:
    - If data don't exist in accelerator's memory, sufficient space is allocated and data transfer from host memory is initiated; the reference count is set to one
    - If data already exist, their reference count is incremented
  - On region exit:
    - The reference count is decremented
    - If reference count is zero, the memory is deallocated

# Data Offload Management (cont.)

- **copyin(*variable-list*)**
  - The data are copied on entry to parallel region
  - Reference counts updated as for the **copy** clause
- **copyout(*variable-list*)**
  - Copies data upon exit from parallel region
  - Other operations (including reference counting) are performed as described for **copy** with the exception of initial data copy from host memory
- **create(*variable-list*)**
  - Creates data structure in accelerator's memory
  - No data transfers between host and device memory are performed
  - Reference counting is handled as described for the **copy** clause

# Supported Array Types

- Statically allocated arrays

```
int cnt[4][500];
```

- Pointers to fixed arrays

```
typedef double vec[1000];
vec *v1;
```

- Statically allocated pointer arrays

```
float *farray[500];
```

- Pointers to array of pointers

```
double **dmat;
```

- Note: *copied data must be contiguous in host memory!*

# Example: Matrix-Vector Multiplication Using copy Clauses

```
001 #include <stdio.h>
002 #include <stdlib.h>
003
004 int main(int argc, char **argv) {
005     unsigned N = 1024;
006     if (argc > 1) N = strtoul(argv[1], 0, 10);
007
008     // create triangular matrix
009     double **restrict m = malloc(N*sizeof(double *));
010    for (int i = 0; i < N; i++)
011    {
012        m[i] = malloc(N*sizeof(double));
013        for (int j = 0; j < N; j++)
014            m[i][j] = (i > j)? 0: 1.0;
015    }
016
017     // create vector filled with ones
018     double *restrict v = malloc(N*sizeof(double));
019     for (int i = 0; i < N; i++) v[i] = 1.0;
020
021     // create result vector
022     double *restrict b = malloc(N*sizeof(double));
023
024     // multiply in parallel
025     #pragma acc kernels copyin(m[:N][:N], v[:N]) copyout(b[:N])
026     for (int i = 0; i < N; i++)
027     {
028         b[i] = 0;
029         for (int j = 0; j < N; j++)
030             b[i] += m[i][j]*v[j];
031     }
032
033     // verify result
034     double r = 0;
035     for (int i = 0; i < N; i++) r += b[i];
036     printf("Result: %f (expected %f)\n", r, (N+1)*N/2.0);
037
038     return 0;
039 }
```

# Loop Scheduling

```
#pragma acc loop [clause-list]
    for (...)
```

- May be specified as a separate directive (shown above) or a clause in *parallel* or *kernels* directive
- Applies to the for loop following immediately
- Accepts a number of additional clauses discussed in the next pages

# Loop Clauses

- **collapse**: determines how many nested loop levels are affected by the scheduling clauses (default: only the nearest loop following the directive)
- **gang**: distributes iteration of the affected loops across available gangs
- **worker**: controls assignment of loop iterations to workers
- **vector**: affects execution of loop iterations in vector or SIMD mode
- **auto**: forces the compiler to analyze data dependencies to come up with parallelization strategy (implied by default in the *kernels* directive)
- **independent**: forces the compiler to treat loop iterations as data independent
- **reduction**: assigns specified scalar variables as a shared reduction variables along with a reduction operation to be carried out at the end of the loop

# Reduction Example

- Code:

```
001 #include <stdio.h>
002 #include <stdlib.h>
003
004 const int N = 10000;
005
006 int main() {
007     double x[N], y[N];
008     double a = 2.0, r = 0.0;
009
010     #pragma acc kernels
011     {
012         // initialize the vectors
013         #pragma acc loop gang worker
014         for (int i = 0; i < N; i++) {
015             x[i] = 1.0;
016             y[i] = -1.0;
017         }
018
019         // perform computation
020         #pragma acc loop independent reduction(+:r)
021         for (int i = 0; i < N; i++) {
022             y[i] = a*x[i]+y[i];
023             r += y[i];
024         }
025     }
026
027     // print result
028     printf("Result: %f (expected %f)\n", r, (float)N);
029
030     return 0;
031 }
```

- Output:

```
Result: 10000.000000 (expected 10000.000000)
```

# Variable Scope

- Depends on the location of their declarations in the code
- Loop variables are considered private to each thread that executes loop iterations
- Variables declared within vector-partitioned code are private to thread associated with each vector lane
- For code executed in worker-partitioned, vector-single mode, the variables are private to each worker, but shared across vector lanes associated with that worker
- Variables declared in a worker-single block are private to the containing gang, but shared across workers and vector lanes in that gang

# Variable Scope Restrictions

- **private clause**
  - Accepts variable names as argument
  - A private copy is created for each parallel gang in *parallel* directive
  - Copies created for thread associated with each vector lane in the *loop* context
- **firstprivate**
  - Used with the *parallel* construct
  - Semantics similar to **private**, but additionally initialized to the variable value stored in the first thread encountering the *parallel* directive

# Atomics Support

```
#pragma acc atomic [atomic-clause]
    statement;
```

- Separate clauses for **read**, **write**, **update**, and **capture**
- **update** clause is assumed by default (i.e., read-modify-update semantics)
- **capture** clause is used to store the original or final value of atomically updated variable

# Atomic Variable Example

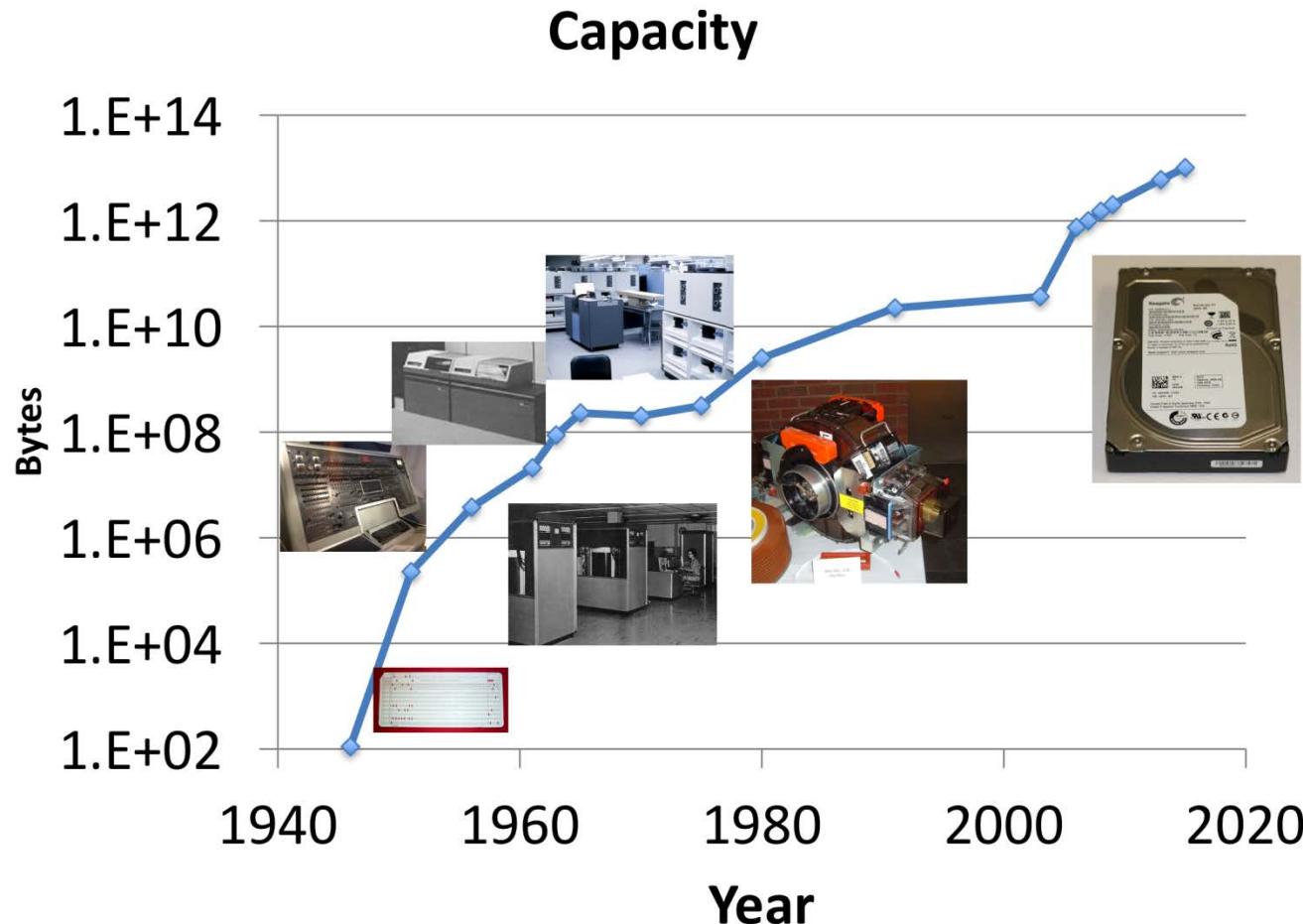
```
001 #include <stdio.h>
002
003 int main(int argc, char **argv) {
004     if (argc == 1) {
005         fprintf(stderr, "Error: file argument needed!\n");
006         exit(1);
007     }
008     FILE *f = fopen(argv[1], "r");
009     if (!f) {
010         fprintf(stderr, "Error: could not open file \'%s\'\n", argv[1]);
011         exit(1);
012     }
013
014     const int BUFSIZE = 65536;
015     char buf[BUFSIZE], ch;
016     // initialize histogram array
017     int hist[256], most = -1;
018     for (int i = 0; i < 256; i++) hist[i] = 0;
019
020     // compute histogram
021     while (1) {
022         size_t size = fread(buf, 1, BUFSIZE, f);
023         if (size <= 0) break;
024         #pragma acc parallel loop copyin(buf[:size])
025         for (int i = 0; i < size; i++) {
026             int v = buf[i];
027             #pragma acc atomic
028             hist[v]++;
029         }
030     }
031     // print the first highest peak
032     for (int i = 0; i < 256; i++) {
033         if (hist[i] > most) {
034             most = hist[i]; ch = i;
035         }
036     printf("Highest count of %d for character code %d\n", most, ch);
037
038     return 0;
039 }
```

# Mass Storage

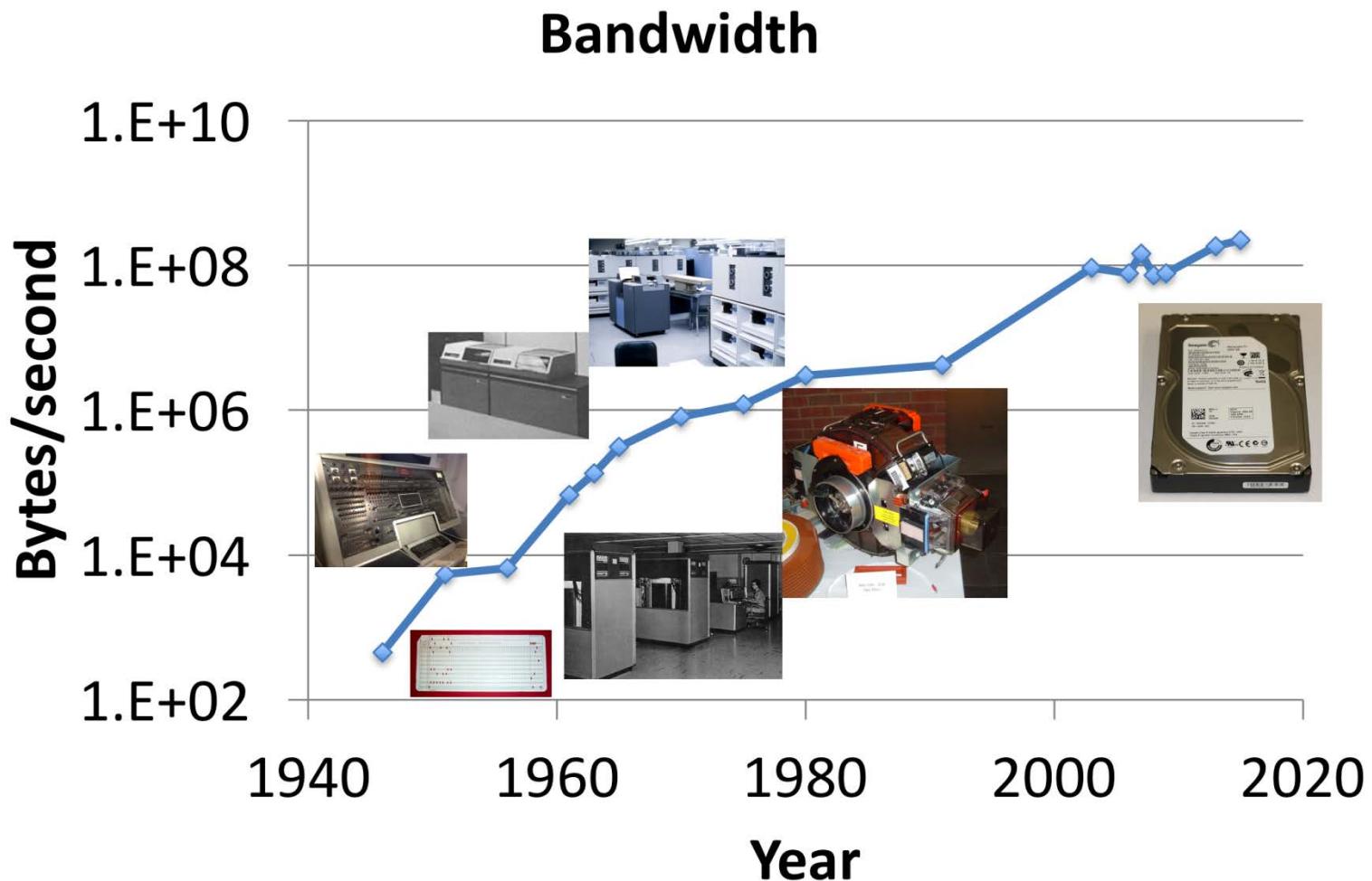
# Overview

- Mass storage enables the computational state retention to be persistent between power cycles of the machine.
- The majority of storage systems utilize four main types of mass storage devices: hard disk drives, solid-state drives, magnetic tapes, and optical storage. Although they serve largely the same purpose, they substantially differ in the underlying physical phenomena used to implement data retention as well their operational characteristics and cost.

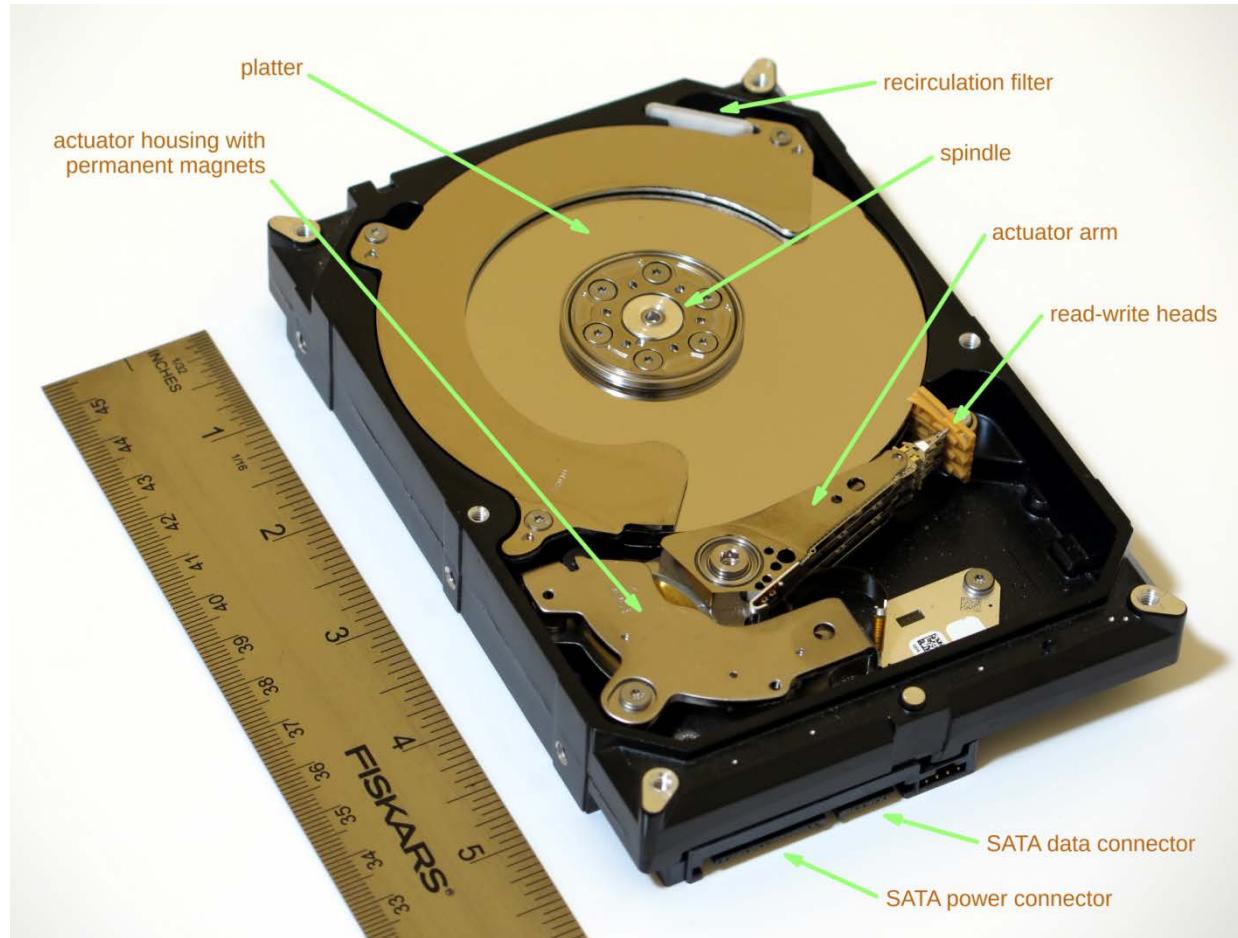
# Brief History of Mass Storage



# Brief History of Mass Storage

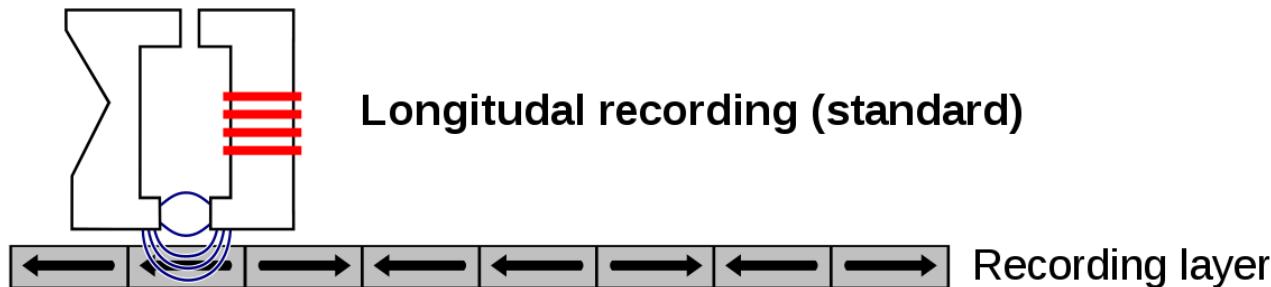


# Internal components of a hard disk drive

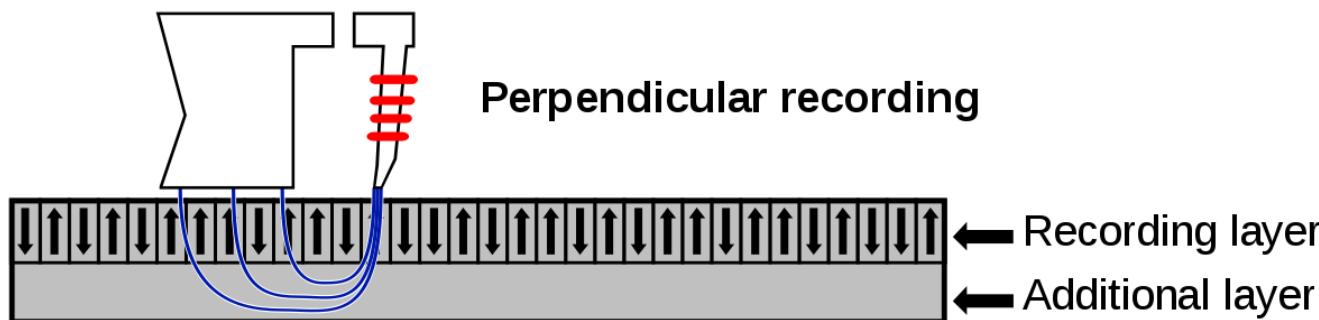


# Bit density increase with perpendicular recording

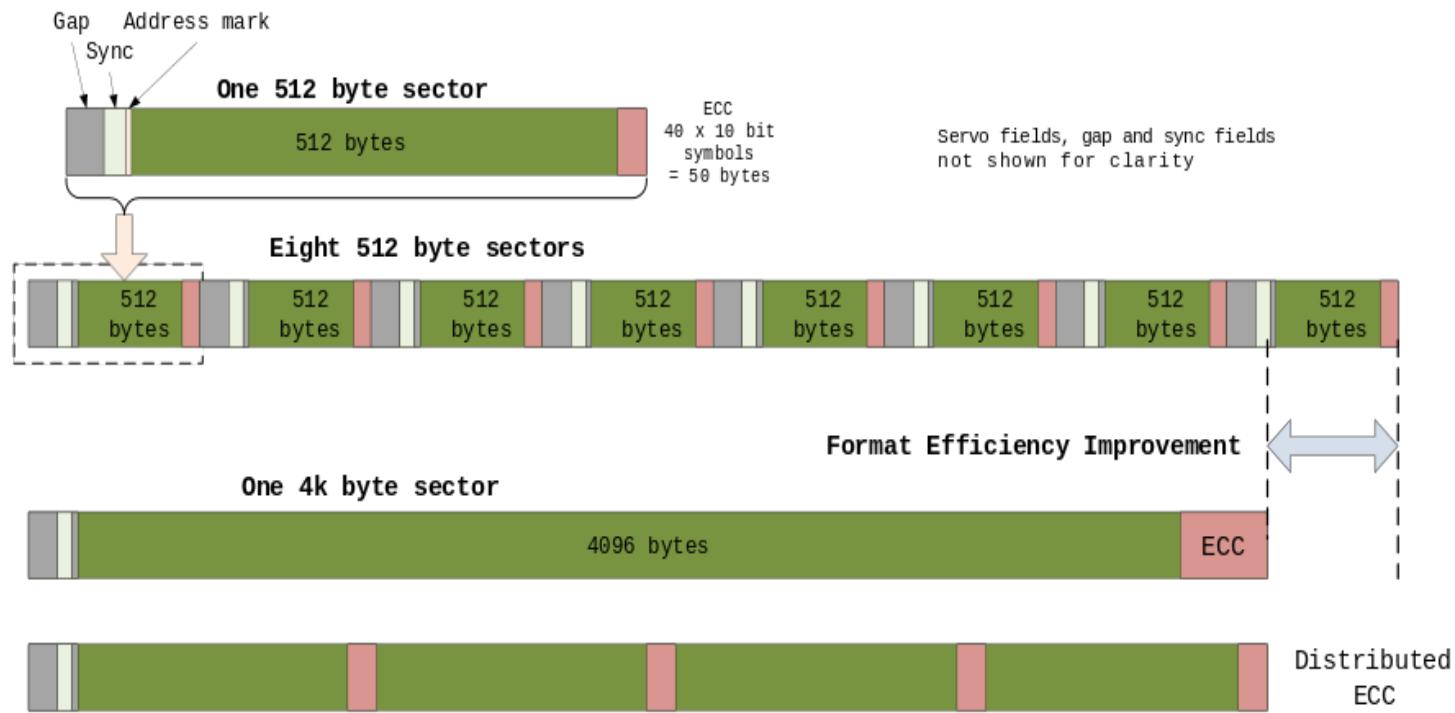
"Ring" writing element



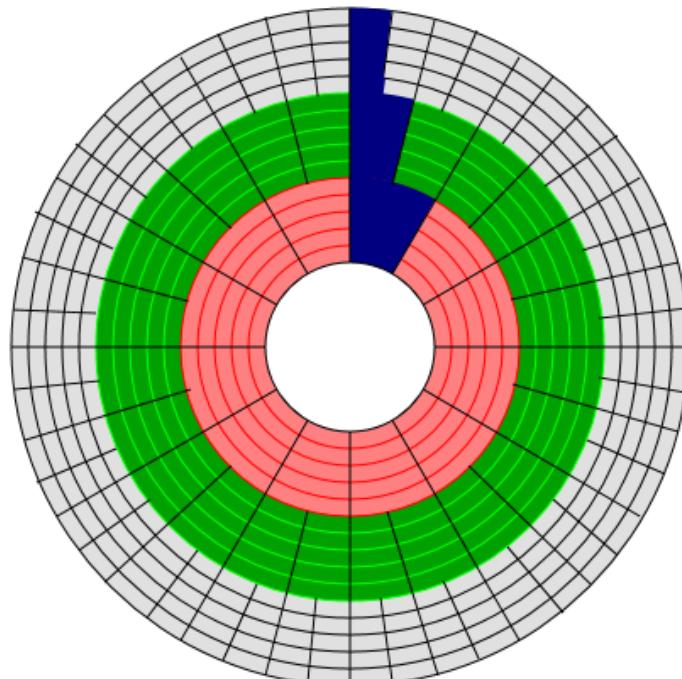
"Monopole" writing element



# Physical information layout on HDDs: advantage of larger sectors



# Physical information layout on HDDs: zone bit recording

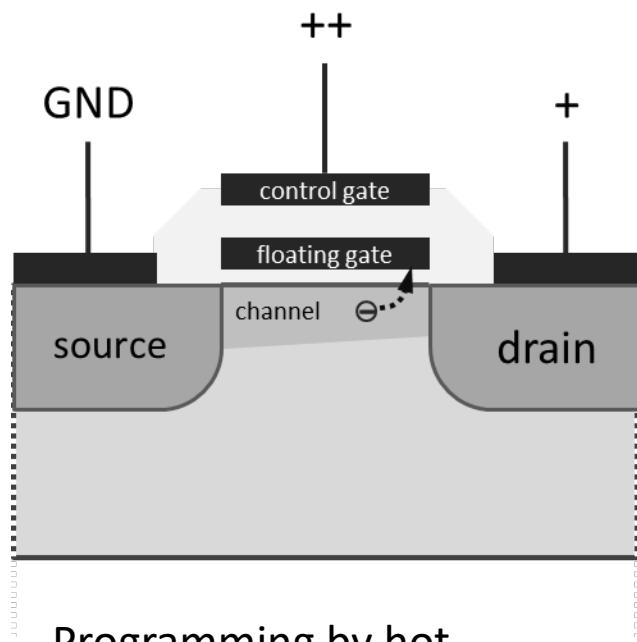


■ Sector 0

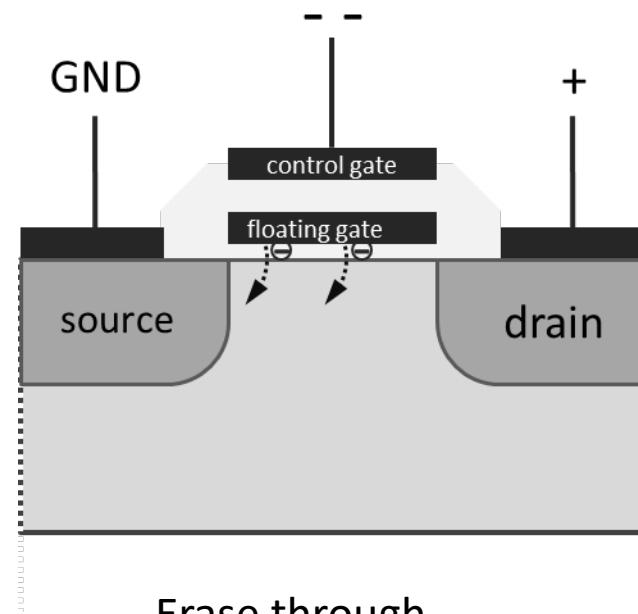
# Comparison of characteristic hard disk drive properties from several manufacturers

Manufacturer and drive	Capacity [TB]	Media transfer rate [MB/s]	Track to track		RPM	Cache (DRAM/flash) [MB]	MTBF [million hours]	Avg. power [W]		UER	Seek	Idle	Acoustic noise [dB(A)]	Form factor [inches]	Market segment
			Seek time [ms]	Full stroke				Seek	Idle						
WDC WD101KRYZ	10	249			7200	256/0	2.5	7.1	5.0	<1 in $10^{15}$	36	20	3.5		Enterprise
WDC WD60EZTZ	6	175			5400	64/0		5.3	3.4	<1 in $10^{14}$	28	25	3.5		Economy desktop
HGST HTS541010A9E680	1	124	1	20	5400	8/0		1.8	0.5		26	24	2.5		Mobile
Seagate ST10000VX0004	10	210				256/0	1	6.8	4.42	<1 in $10^{15}$			3.5		A/V streaming
Seagate ST2000DX002	2	156	<9.5 (average)			64/8192		6.7	4.5	<1 in $10^{14}$			3.5		Performance desktop

# FGMOS Transistor

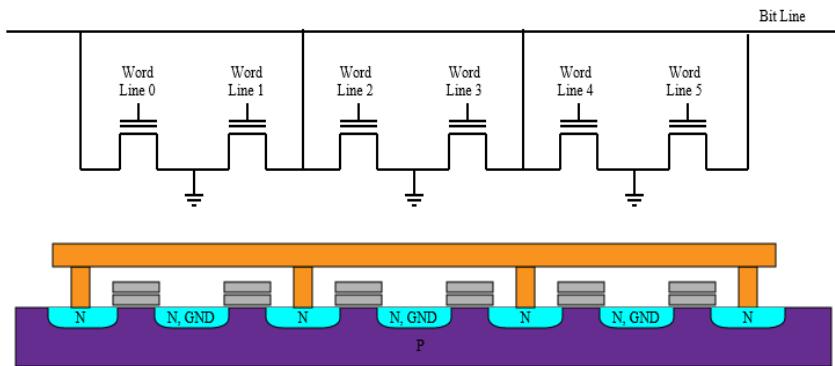


Programming by hot  
electron injection

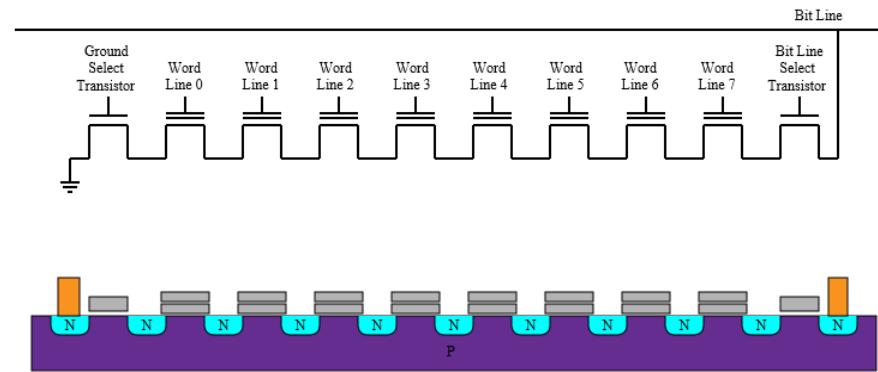


Erase through  
quantum tunneling

# Storage cell connections and corresponding hardware implementation



NOR flash memory



NAND flash memory

# Comparison of principal properties of NOR and NAND flash memory

Property	NOR flash	NAND flash
Capacity	Low	High
Cost per bit	High	Low
Read speed	High	Medium
Write speed	Very slow	Slow
Erase speed	Very slow (10s to 100s of ms)	Medium (single ms)
Erase cycles (endurance)	100,000 – 1,000,000	1,000 – 10,000
Active power	High	Low
Standby power	Low	Medium
Random access	Easy	Hard
Block storage	Medium	Easy

# Examples of currently manufactured SSD devices and their operational properties

Manufacturer and device	Capacity [GB]	Sequential read [MB/s]	Sequential write [MB/s]	Max. 4KB random reads [kIOPS]	Max. 4KB random writes [kIOPS]	Terabytes written	MTTF [million hours]	Power (active/idle) [W]	Memory type	Interface
Crucial <a href="#">CT2050MX30 0SSD1</a>	2,050	530	510	92	83	400	1.5	0.15 (avg.)	3D TLC NAND	SATA 6Gbps
Samsung MZ-V6P2T0BW	2,048	3,500	2,100	440	360	1,200		(5.8/1.2)	48-layer MLC V-NAND	NVMe 1.1, PCIe 3.0 x4
SanDisk SDFADCMOS-6T40-SF1	6,400	2,800	2,200	285	385	22,000		25 (peak)	MLC	PCIe 2 x8

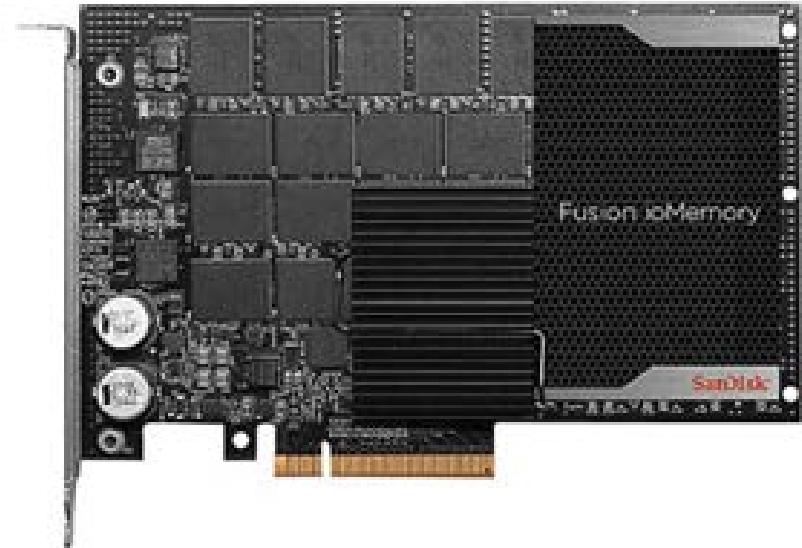
# SSD Examples



Crucial MX300 series  
(2.5" form factor)



Samsung 960 PRO  
series (M.2 form factor)



SanDisk Fusion ioMemory SX350 series (8-lane, PCI-Express card)

# Advances in magnetic tape storage



IBM 726 from 1951



IBM 3480 format tape



IBM 3480 deck subsystem from 1984 with an older 3480 system in the background

# Comparison of dominant tape storage families



DDS-1 (1989)



DLT-IV (1999)

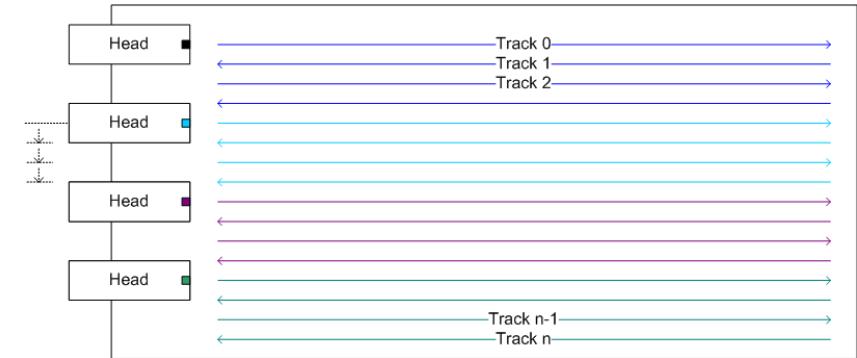


LTO-2 (2005)

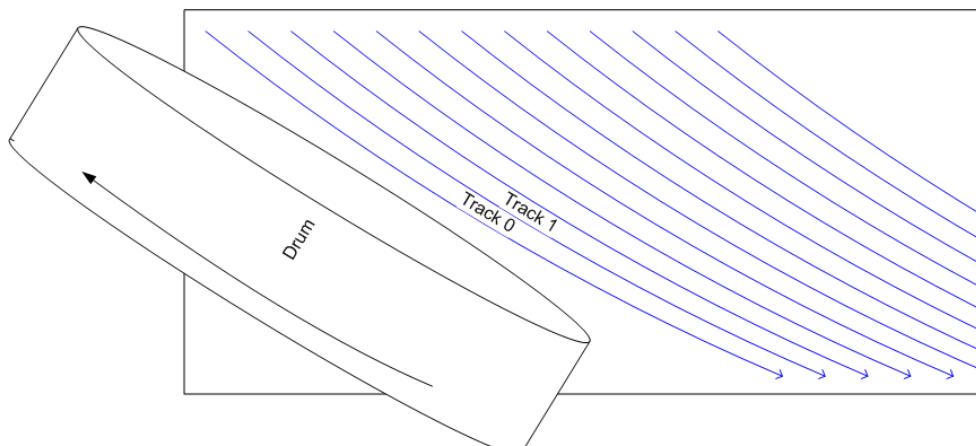
# Tape recording formats



Linear



Linear-serpentine



Helical

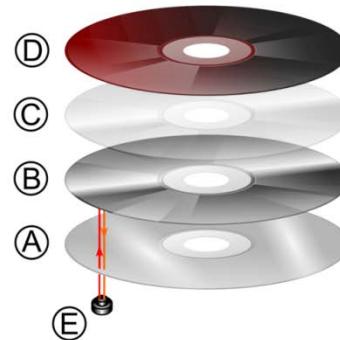
# Operational parameters of selected tape drives

Manufacturer and drive	Capacity [TB]	Sustained data rate [MB/s]	High-speed search [m/s]	Max. operating power [W]	Data format	Cartridge types supported	Interface
IBM TS1150	Up to 10 (medium dependent)	360, 300	12.4	46	32-channel linear-serpentine	IBM 3592 Gen. 3 and 4	8Gbps fibre channel
HP Enterprise BB873A	Up to 6	300			32-channel linear-serpentine	LTO-7 (RW), LTO-6 (RW), LTO-5 (RO)	6Gbps SAS

# Optical Storage: Compact Disc

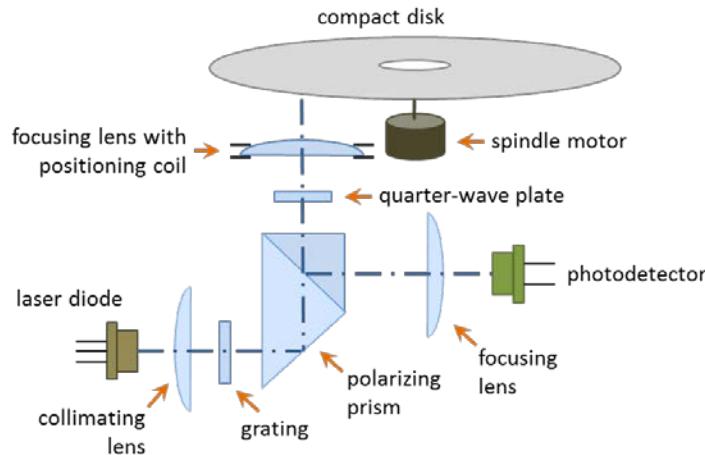


Medium

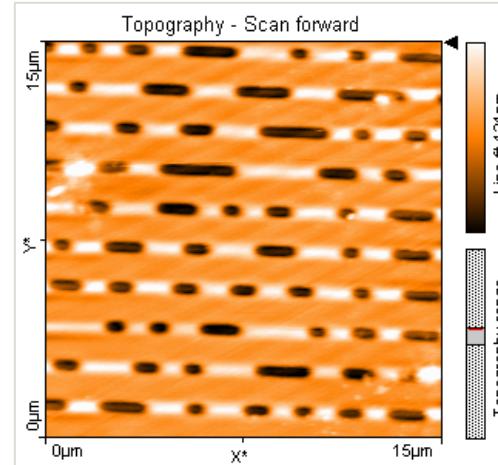


- A) Transparent polycarbonate layer with data encoded in pits
- B) Reflective metallic layer
- C) Protective lacquer layer
- D) Disc label
- E) Laser beam

Component layers

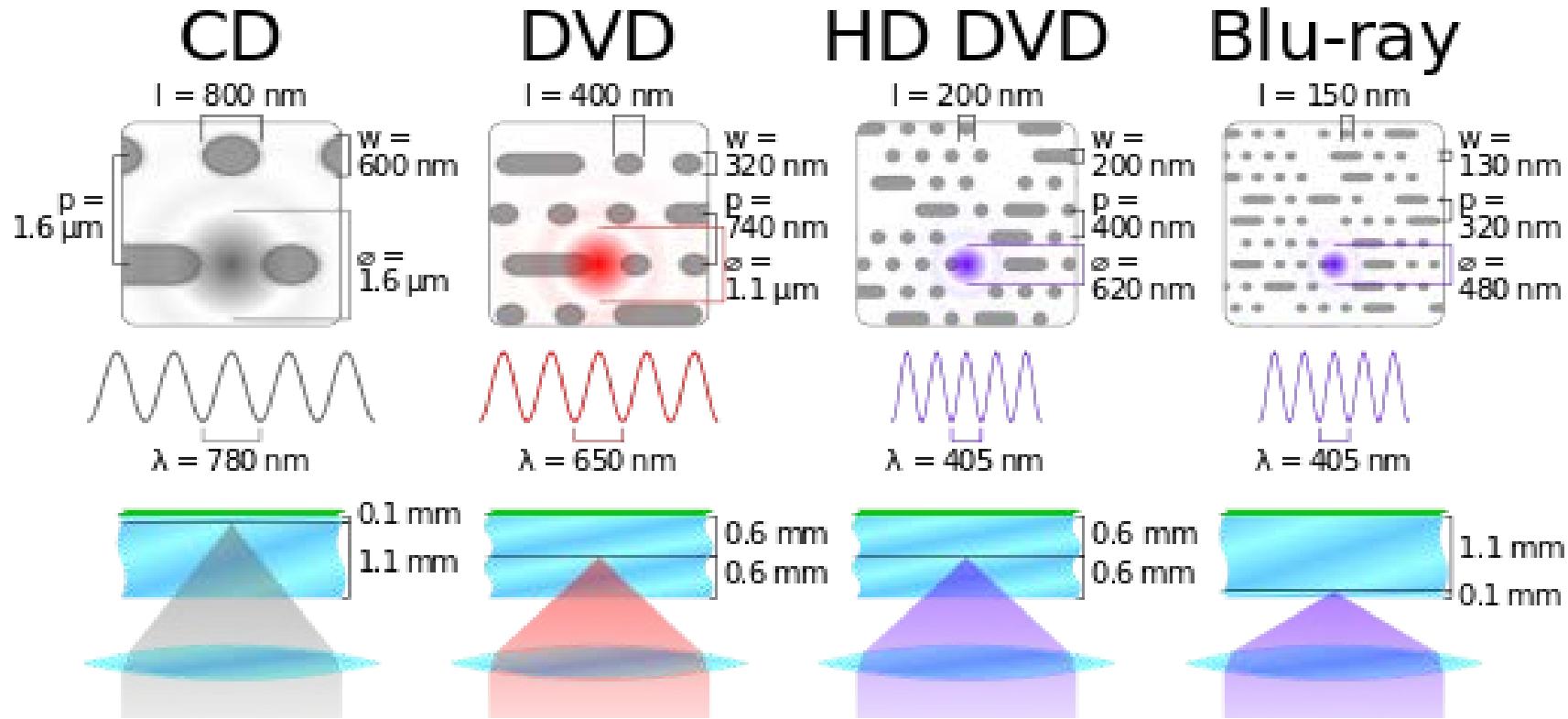


Optical pickup mechanism



Geometric properties of the data track

# Comparison of optical format geometries (CD, DVD, HD-DVD, Blu-ray)

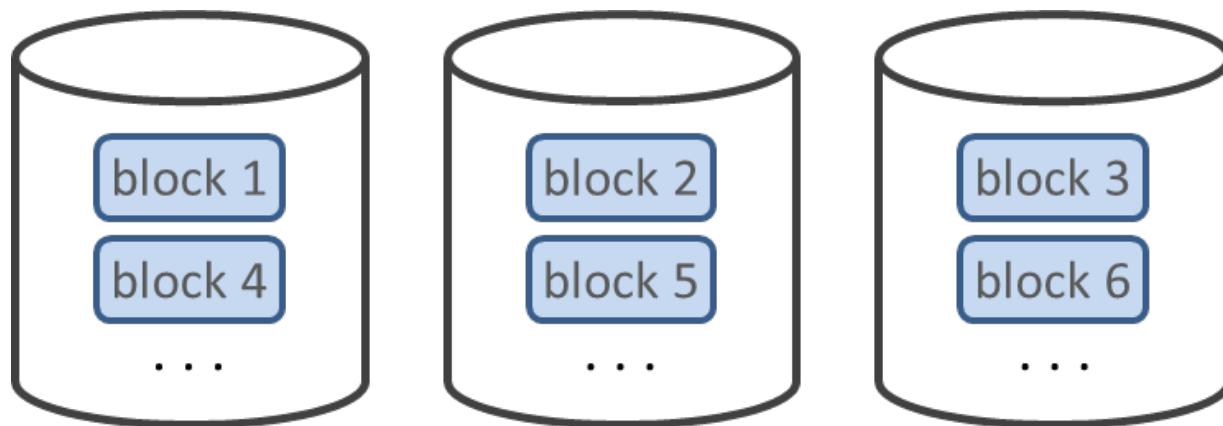


Listed parameters denote minimum feature length ( $l$ ), track width ( $w$ ), track pitch ( $p$ ), laser beam diameter ( $\phi$ ), and wavelength ( $\lambda$ )

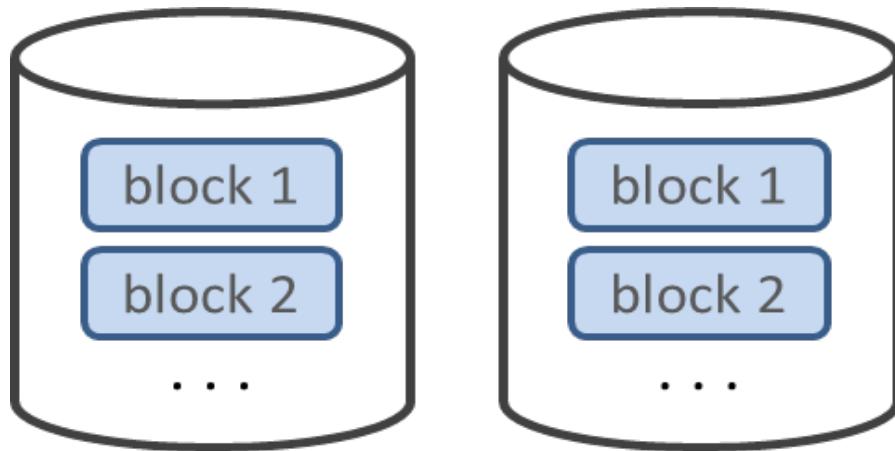
# Parameters of a typical consumer-grade multi-format optical drive

Manufacturer and drive	BD access time [ms]	DVD access time [ms]	CD access time [ms]	Max. Data Rate						Buffer size [MB]	Interface
				BD read	BD write	DVD read	DVD write	CD read	CD write		
Lite-On iHBS312	250 (SL) 380 (DL)	150 (ROM) 160 (DL) 200 (RAM)	150	6× (RE DL) 8×(SL)	2× (rewrite) 8× (DL) 12× (SL)	16×	6× (rewrite) 12× (RAM) 16× (+R, -R)	48×	24× (-RW) 48× (-R)	8	SATA (internal)

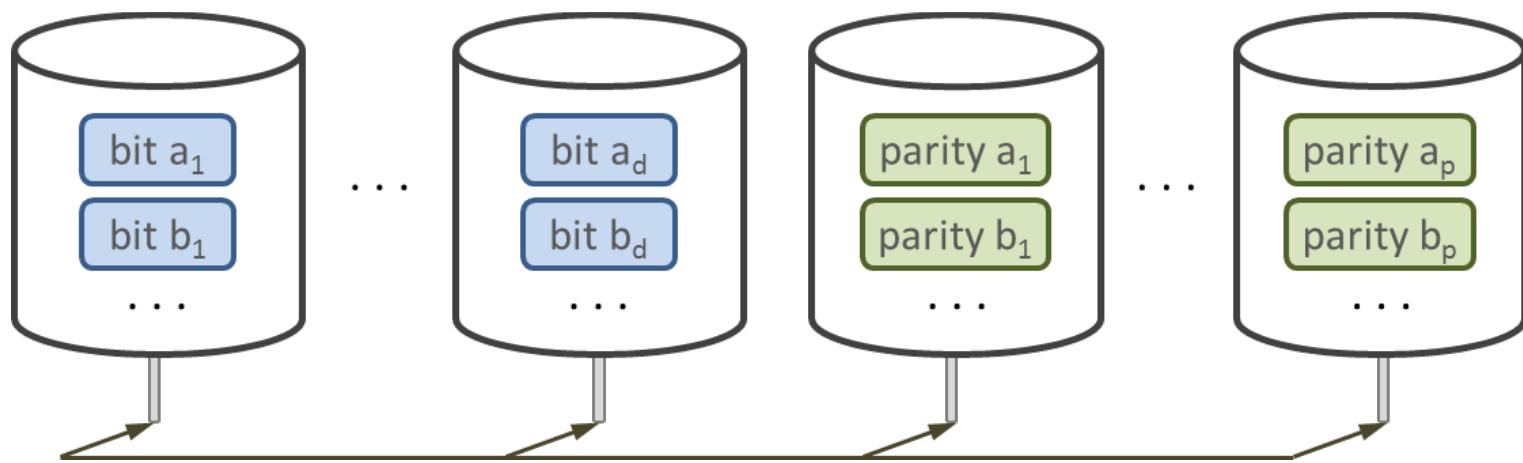
# RAID 0: striping



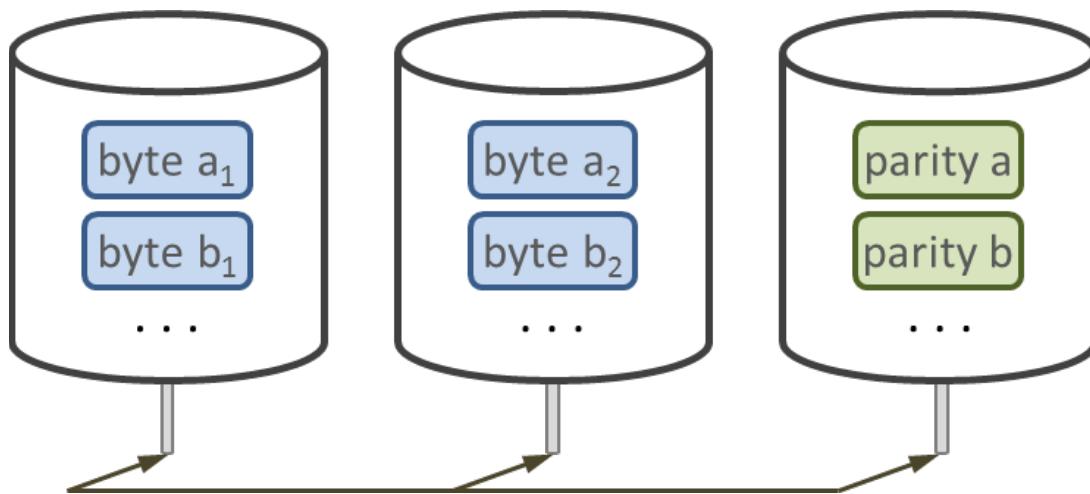
# RAID 1: Mirroring



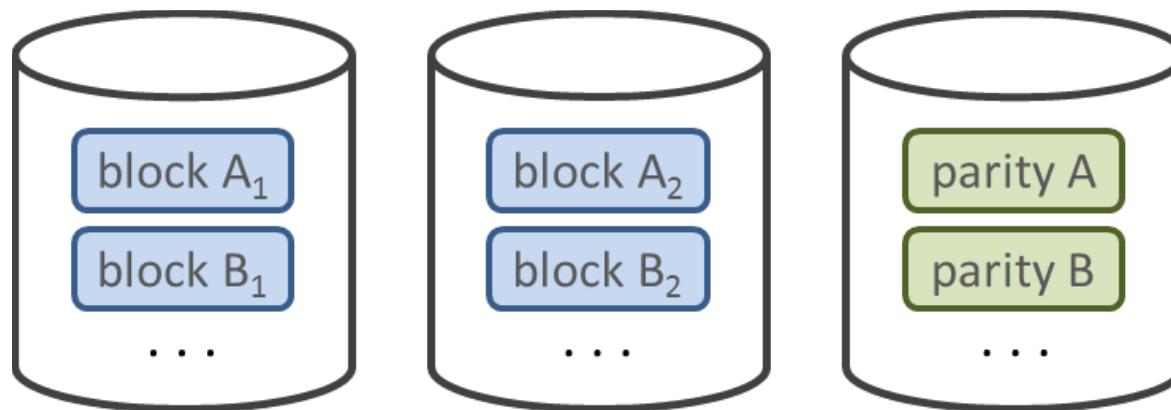
# RAID 2: bit-level striping with Hamming code



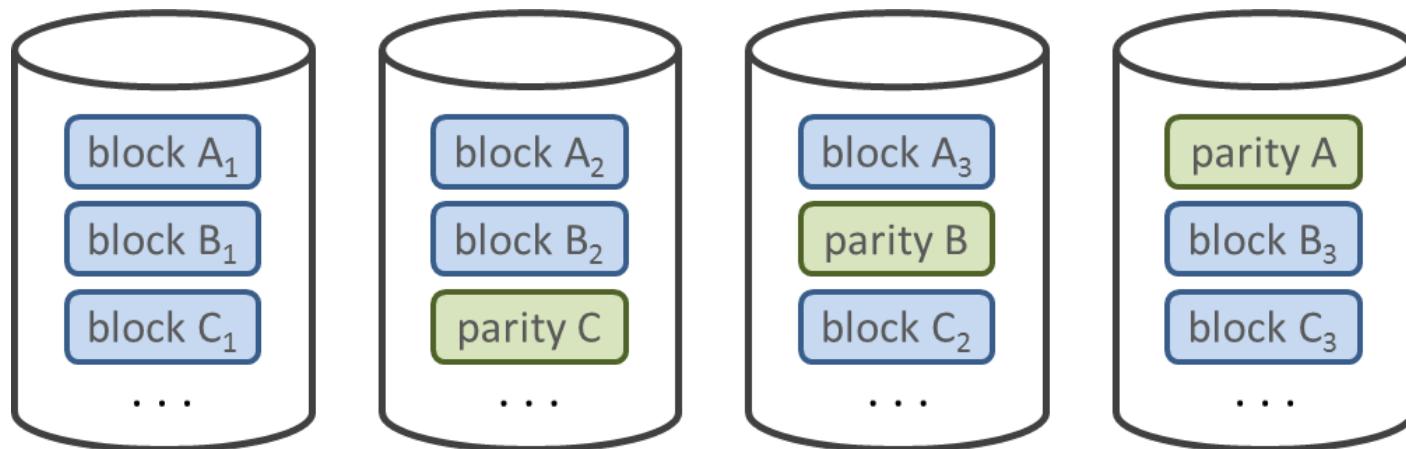
# RAID 3: byte-level striping with dedicated parity



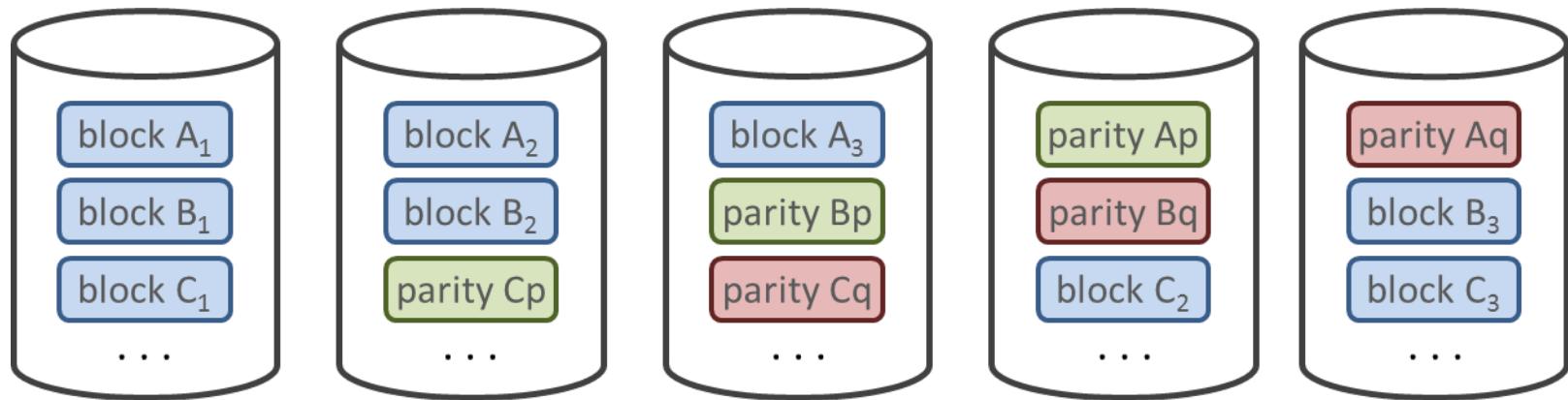
# RAID 4: block-level striping with dedicated parity



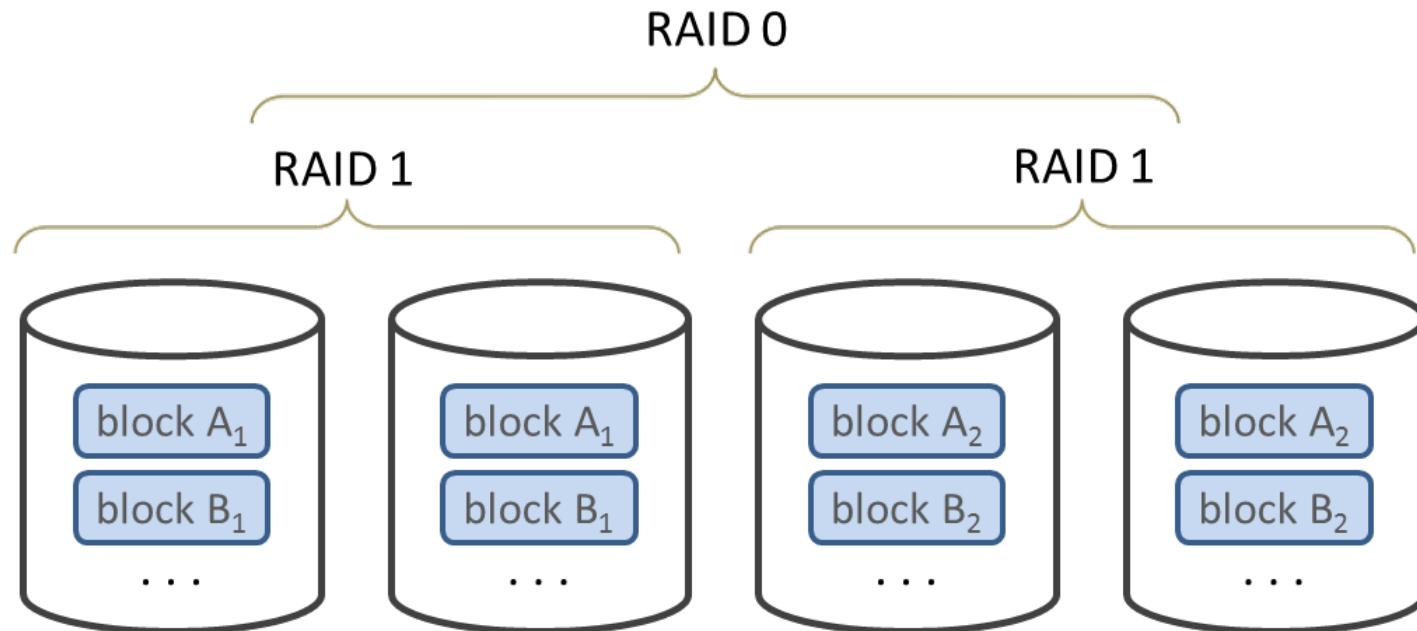
# RAID 5: block-level striping with single distributed parity



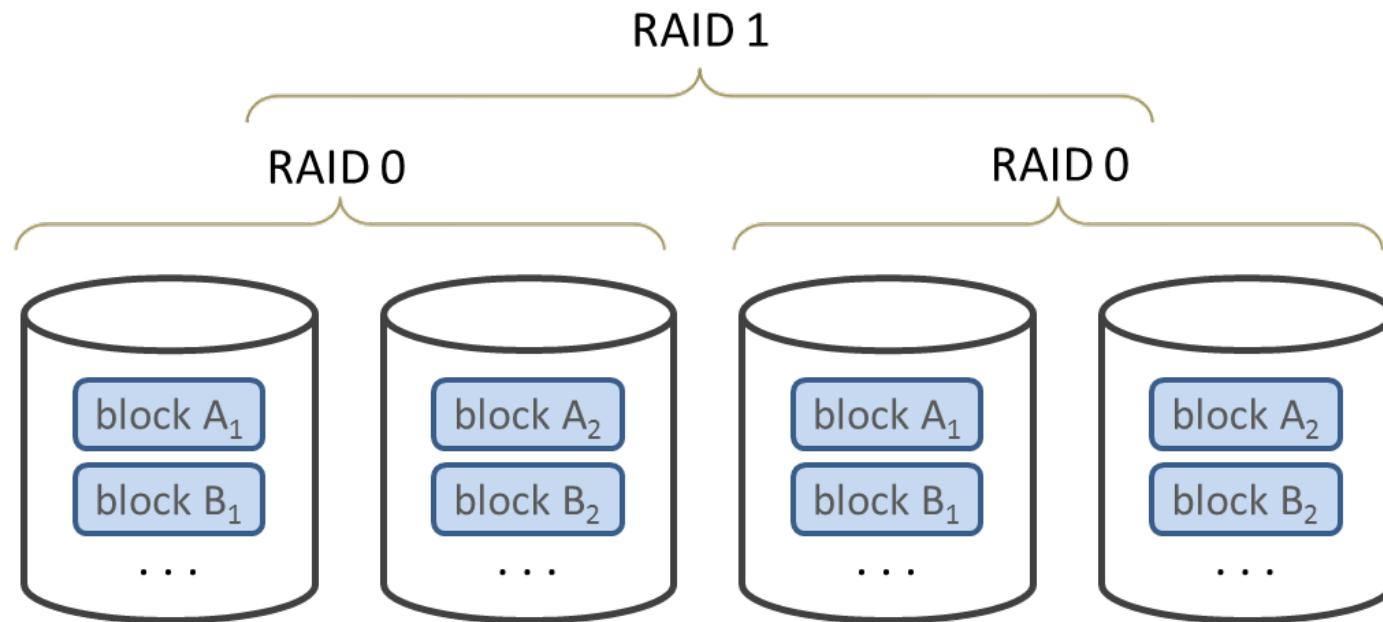
# RAID 6: block-level striping with dual distributed parity



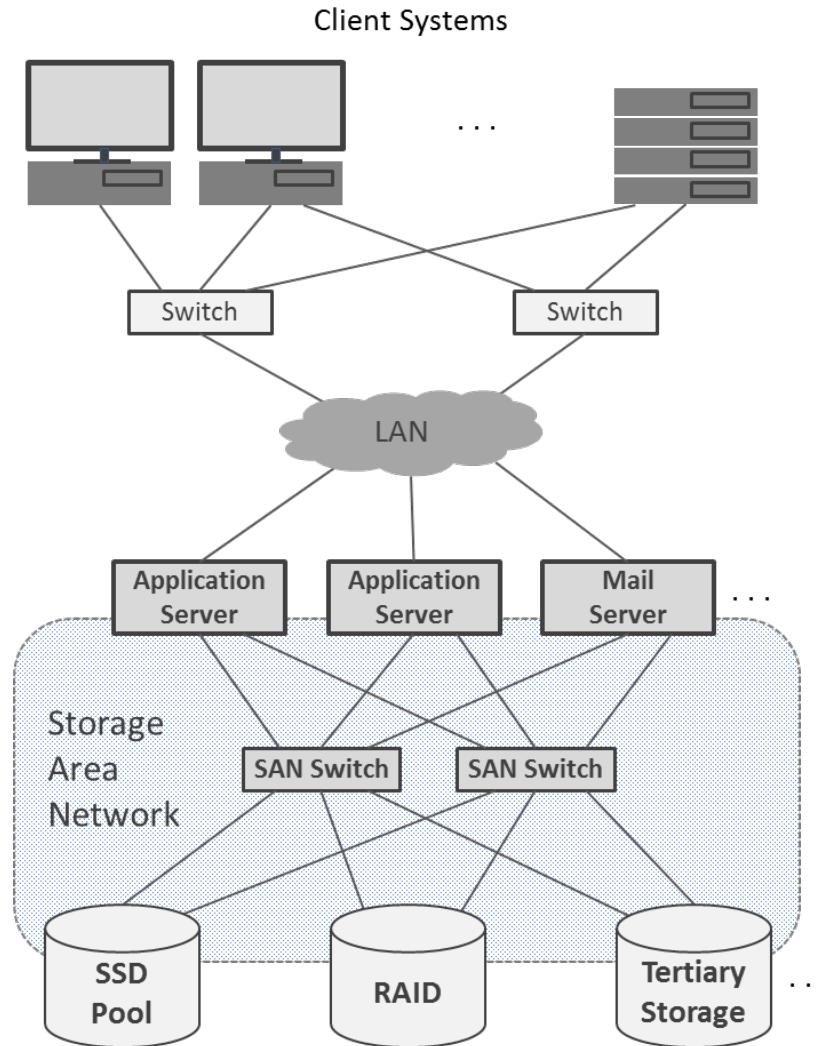
# Hybrid RAID variants: RAID 10 (stripe of mirrors)



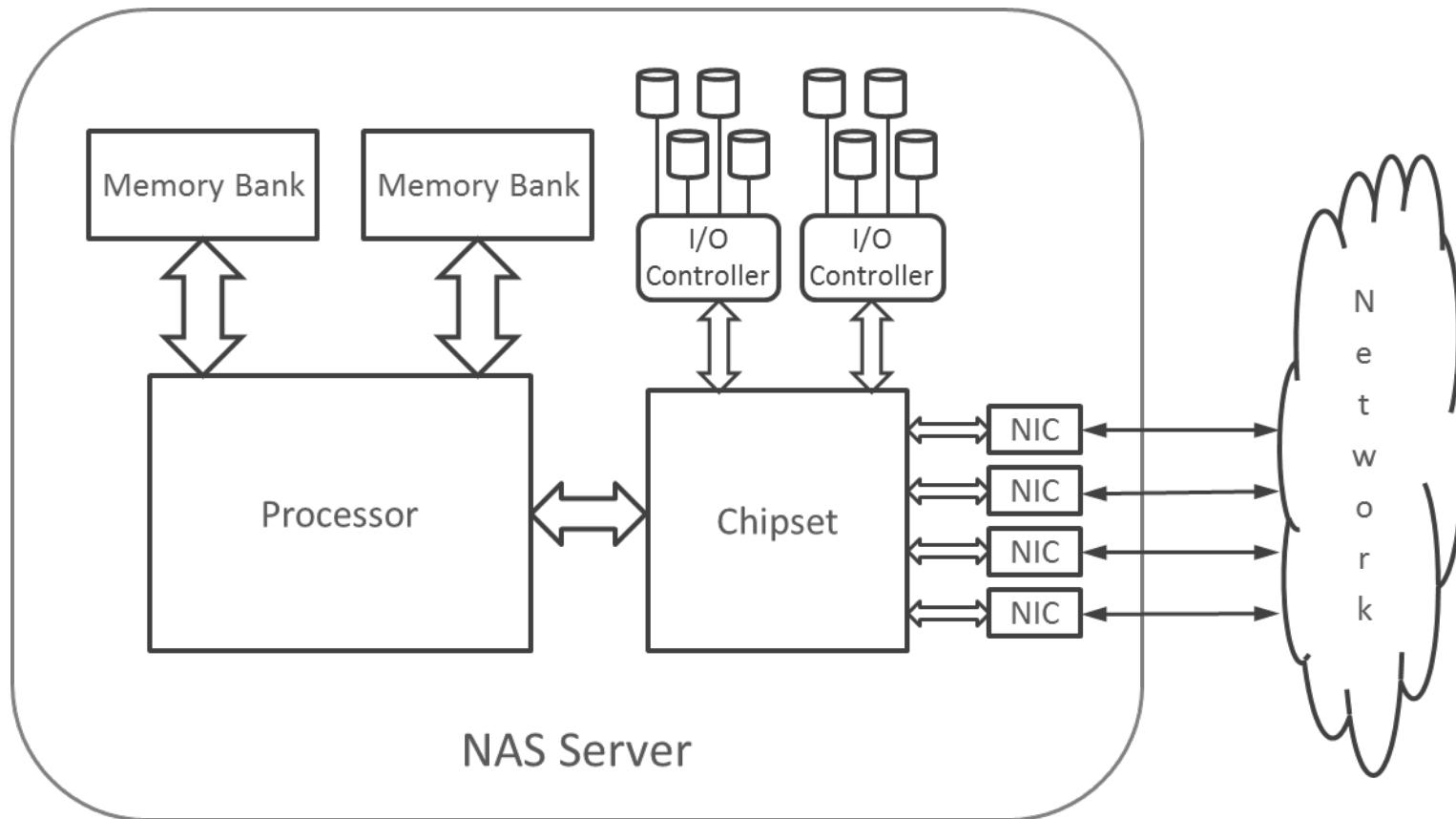
# Hybrid RAID variants: RAID 01 (mirror of stripes)



# Example configuration of Storage Area Network



# Simplified architecture of NAS server



# Properties of selected high-capacity tertiary storage systems

Manufacturer	Product	Max. capacity [TB]	Media slots	Media type	Interface	Power
Quantum	Scalar i6000 tape library	180,090	12,006	LTO-7 cartridge	8 Gbps Fibre Channel	24 kVA
HIT Storage	HMS-5175 BluRay library	175	1,750	100 GB BDTL disc	1 Gbps Ethernet	

# Tertiary storage platforms



Quantum tape library



BluRay optical jukebox

# File Systems

# Introduction

- Large amounts of data need to be stored and managed in HPC systems
  - Different types of I/O devices
  - Different name space
  - Different interfaces
- Providing uniform view of shared data in distributed environments
- Coordination of concurrent and parallel accesses
- Examples:
  - POSIX File API
  - Network File System (NFS)
  - General Parallel File System (GPFS)/Spectrum Scale
  - Lustre parallel file system

# Abstracting Device Level Access

- Storage devices are block devices
  - Data stored in physical blocks (*a.k.a.* sectors or records)
  - Access granularity ranges from 512 bytes to 16 KB
  - Low level access via physical addresses with potentially different translation algorithms for different devices
- Performance considerations
  - Read vs. write (e.g., optical disks)
  - Access ordering (optimization of head movement)
  - In-memory caching support
- Resilience considerations
  - Mitigation of media faults impact (“bad blocks”)
  - Reliability of multiple-device storage pools
- Need for unified interface
  - Programmer’s productivity

# File System Functions (I)

- Storage and management of metadata (additional attributes associated with the actual data)
  - Ownership, Access Control Lists, time of creation/access/modification, setuid/setgid, “sticky bits”, extended attributes, ...
- Organization of data
  - Hierarchy of directories and files
  - Files contain data
  - Directories store files and other directories
- Namespace implementation
  - Each directory and file has a name (not necessarily unique)
  - Individual components are identified by paths:
    - Paths originate in file system’s root
    - Contain all parent component names separated by a predefined character (e.g., “/”)
  - Paths uniquely identify each component (file or directory)
  - Other constructs: links, drive letters, ...

# File System Functions (II)

- Storage space management
  - Physical space allocation
  - Maximization of performance for anticipated access patterns
    - Contiguous blocks within HDD tracks
    - Adjacent tracks to minimize HDD head movement latency
  - Load leveling (SSDs)
  - Session management with optical disks
- File system mounting
  - Import of namespaces on other devices
- Special file support
  - Sockets, named pipes, raw devices
  - Additional functionality available via *fcntl* and *ioctl* interfaces
- Fault handling
  - Preserves data integrity
  - Sources of faults and inconsistencies
    - Media errors
    - Device errors
    - I/O interface errors
    - Power fluctuations
    - State replication (e.g., file buffer cache in memory)

# Types of File Systems

- Optimized for storage medium
  - HDDs (Extfs, Btrfs, ZFS, JFS, XFS, HFS+, VFAT, NTFS, ...)
  - SSDs (ExtremeFFS, TrueFFS, also: kernel extensions of common HDD file systems)
  - Flash memories (JFFS2, F2FS, UBIFS)
  - Optical storage (ISO 9660, UDF)
  - RAM “disks” (ramfs, tmpfs, ...)
  - ROM (CramFS, Romfs, SquashFS)
- Distributed (AFS, NFS, )
- Parallel (PVFS2, OrangeFS, GPFS, Lustre, PanFS, ...)
- Pseudo-file systems (procfs, sysfs, devfs, debugfs)
- Many others

# File System Features of Interest

- Design limits
  - Maximum file size
  - Total number of files
  - Total number of directories
  - Total number of files per directory
  - Name and path length restrictions
- In-memory caching of data and metadata
  - Write-back
  - Write-through
  - None
- Integrity mechanisms
  - Journaling (metadata and/or block level)
  - Parity based mechanisms
- Storage and performance optimizations
  - Log-structured (fast writes)
  - Copy-on-write
  - Extent support
  - Sparse file support
  - Variable block size
  - Tail merging
  - Execute-in-place
- Volume resizing
  - Growing and shrinking
  - On- vs. off-line
- Access rights
  - Unix-like
  - ACL
  - Capability based
- Transparent data compression
- Data de-duplication
- Data encryption
  - Partition/volume-wide
  - Per file
- Distributed operation
  - Distributed storage servers
  - Distributed client support
  - Concurrent access to shared data
  - Network type and protocol support
- Availability for specific platform and OS

# POSIX File Access Interface

- “Portable Operating System Interface”
- IEEE standard 1003.1 (originated in 1988)
- File and directory API a part of specification
- Two flavors
  - System call based
    - Higher overhead due to crossing into kernel space
    - Poor performance when many calls reading or writing small amounts of data are issued
    - Uses integer descriptors to identify open files
  - Streaming (buffered) I/O
    - Library based with OS backend
    - Transparent data buffers in application space used to limit the number of system calls
    - Uses pointer to FILE structure to describe an open file
    - More portable as a part of ISO/IEC C language standard
    - Function names typically begin with “f”
- Both APIs maintain an implicit file offset (or “pointer”) specifying the location in file at which the operation is performed

# File Open and Close

- System calls

```
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags, ...);
```

Opens a file in desired access mode returning its descriptor or -1 on error. *flags* include or-ed combination of O\_RDONLY, O\_WRONLY, O\_RDWR, O\_APPEND, O\_CREAT, O\_EXCL, O\_TRUNC. The optional third argument specifies access mode if the file creation (O\_CREAT) is requested.

```
#include <unistd.h>
int close(int fd);
```

Closes an opened file returning zero on success or -1 on error.

- Streaming I/O

```
#include <stdio.h>
FILE *fopen(const char *restrict path, const char *restrict mode);
```

Opens a stream in accordance with *mode* parameter returning a pointer to FILE structure on success (or a NULL pointer otherwise). *mode* is either “r”, “w”, or “a” for read access, write access with initial truncation to zero length or append (write at the end of file), respectively. The mode character may be optionally followed by “+” to enable update mode (reading and writing in any order); for “a+” reading is initially performed at the start of the file while write appends to the end of the file.

```
int fclose(FILE *stream);
```

Closes *stream* returning zero on success or EOF is an error has occurred.

# Data Read and Write

- System calls

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, void *buf, size_t n);
```

The *read* call copies *n* bytes from file identified by descriptor *fd* to buffer *buf*. The *write* call writes the contents of *buf* to a file identified by *fd*. Both return the number of bytes transferred or a negative number if an error occurred. The implicit file offset is updated by the number of bytes transferred.

- Streaming I/O

```
#include <stdio.h>
size_t fread(void *restrict buf, size_t item, size_t count, FILE
*restrict stream);
size_t fwrite(void *restrict buf, size_t item, size_t count, FILE
*restrict stream);
```

The calls transfer an integral *count* of elements, each of size *item* bytes, between data buffer *buf* and *stream*. The calls return the actual number of items transferred. It may be less than *count* if the end of file was encountered or an error occurred. The internal file pointer is increased by the number of bytes copied.

# File Offset Manipulation

- System calls

```
#include <unistd.h>
off_t lseek(int fd, off_t offs, int whence);
```

*lseek* updates the internal file offset depending on the value of parameter *whence*:

- SEEK\_SET – the file offset is set to *offs*,
- SEEK\_CUR – the value of *offs* is added to file offset,
- SEEK\_END – the file offset is set to the sum of file size and *offs*.

File offset may be set to point beyond the original end of file; the unwritten portion of file will read as zeroes until explicitly overwritten. The call returns the updated offset value.

- Streaming I/O

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

The *fseek* call sets the file offset for *stream*; the meaning of *whence* argument is the same as for *lseek*. It returns zero on success or -1 on error. *ftell* returns the current value of file offset or -1 on error.

# Data Access with Explicit Offset

- System calls

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t n,
off_t offs);
ssize_t pwrite(int fd, void *buf, size_t n,
off_t offs);
```

The calls have analogous semantics to *read* and *write*, except they are performed at the file offset specified by *offs*. The value of the internal file offset associated with *fd* is not modified.

- Streaming I/O  
No equivalent.

# Data Buffer Flush

- System calls

```
#include <unistd.h>
off_t fsync(int fd);
```

Transfers all data and metadata associated with *fd* to permanent storage. The call blocks until all buffered data are written (zero return value) or an error occurs (return value -1).

- Streaming I/O

```
#include <stdio.h>
int fflush(FILE *stream);
```

Flushes all unwritten data associated with specific *stream* to the underlying OS file (but not necessarily to storage). If the stream was opened for reading, it updates the underlying file's offset to match the current offset of *stream*. If *stream* is NULL, the buffers of all streams opened by the application are flushed. Returns zero on success or EOF on error.

# File Status Query

- System calls

```
#include <fcntl.h>
#include <sys/stat.h>
int lstat(const char *restrict path, struct stat
*restrict buf);
int fstat(int fd, struct stat *restrict buf);
```

The calls retrieve metadata of a file identified by path (*lstat*) or the opened descriptor (*fstat*) and place them in buffer *buf*. The *stat* structure stores, among others, size of file in bytes (*st\_size*), access mode (*st\_mode*), owner's user ID (*st\_uid*), owner's group ID (*st\_gid*), last access time (*st\_atim*), last modification time (*st\_mtim*), and last status change time (*st\_ctim*). Both return zero if the call was successful or -1 on failure.

- Streaming I/O  
No equivalent.

# File Descriptor and Stream Conversion

- Open a stream associated with a file identified by *fd*:

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
```

The *mode* parameter must be compatible with the access type with which the underlying file was opened. “w” and “w+” modes do not truncate the file. File descriptor will be closed when the related stream is closed.

- Retrieve descriptor of a file used by *stream*:

```
#include <unistd.h>
int fileno(FILE *stream);
```

# Example: system calls

```
001 #include <stdio.h>
002 #include <stdlib.h>
003 #include <unistd.h>
004 #include <sys/stat.h>
005 #include <fcntl.h>
006
007 #define BUFFER_SIZE 4096
008 #define HALF (BUFFER_SIZE/2)
009
010 int main(int argc, char **argv)
011 {
012     // initialize buffer
013     int wbuf[BUFFER_SIZE], i;
014     for (i = 0; i < BUFFER_SIZE; i++) wbuf[i] = 2*i+1;
015
016     // open file, write buffer contents, and flush it to the storage
017     int fd = open("test_file.dat", O_WRONLY|O_CREAT|O_TRUNC, 0600);
018     int bytes = BUFFER_SIZE*sizeof(int);
019     if (write(fd, wbuf, bytes) != bytes) {
020         fprintf(stderr, "Error: truncated write, exiting!\n");
021         exit(1);
022     }
023     fsync(fd);
024     close(fd);
025
026     // retrieve the second half of the file and verify
027     int rbuf[HALF];
028     fd = open("test_file.dat", O_RDONLY);
029     bytes /= 2;
030     if (pread(fd, rbuf, bytes, bytes) != bytes) {
031         fprintf(stderr, "Error: truncated read, exiting!\n");
032         exit(1);
033     }
034     close(fd);
035
036     for (i = 0; i < HALF; i++)
037         if (wbuf[i+HALF] != rbuf[i]) {
038             fprintf(stderr, "Error: retrieved data is invalid!\n");
039             exit(2);
040         }
041     printf("Data verified.\n");
042
043     return 0;
044 }
```

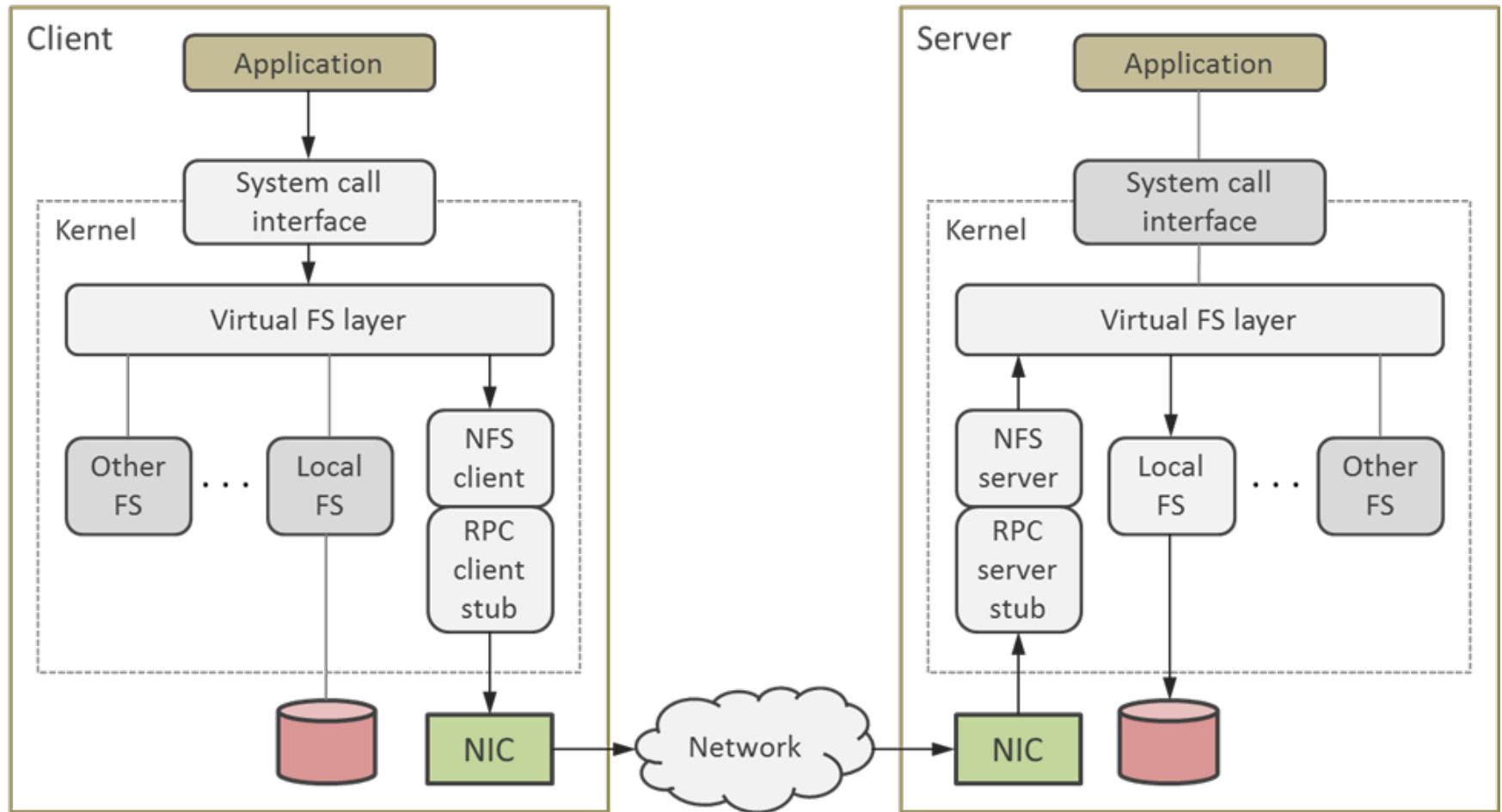
# Example: streams

```
001 #include <stdio.h>
002 #include <stdlib.h>
003 #include <unistd.h>
004
005 #define BUFFER_SIZE 4096
006 #define HALF (BUFFER_SIZE/2)
007
008 int main(int argc, char **argv)
009 {
010     // initialize buffer
011     int wbuf[BUFFER_SIZE], i;
012     for (i = 0; i < BUFFER_SIZE; i++) wbuf[i] = 2*i+1;
013
014     // open file, write buffer contents, and flush it to the storage
015     FILE *f = fopen("test_file.dat", "w");
016     size_t count = BUFFER_SIZE;
017     if (fwrite(wbuf, sizeof(int), count, f) != count) {
018         fprintf(stderr, "Error: truncated write, exiting!\n");
019         exit(1);
020     }
021     fflush(f); fsync(fileno(f));
022     fclose(f);
023
024     // retrieve the second half of the file and verify
025     int rbuf[HALF];
026     f = fopen("test_file.dat", "r");
027     count /= 2;
028     fseek(f, count*sizeof(int), SEEK_SET);
029     if (fread(rbuf, sizeof(int), count, f) != count) {
030         fprintf(stderr, "Error: truncated read, exiting!\n");
031         exit(1);
032     }
033     fclose(f);
034
035     for (i = 0; i < HALF; i++)
036         if (wbuf[i+HALF] != rbuf[i]) {
037             fprintf(stderr, "Error: retrieved data invalid!\n");
038             exit(2);
039         }
040     printf("Data verified.\n");
041
042     return 0;
043 }
```

# Network File System (NFS)

- Developed at Sun Microsystems
- Currently an open standard
- Very broad deployment
- Compatible with POSIX file interface
- Any subdirectory on host may be “exported” to permit remote access
- Works with most lower level file systems
- Clients mount the exported file system anywhere in local directory hierarchy
- Initially used stateless protocol
- Session semantics (limited file sharing, especially of writes)
- May use both TCP/IP and UDP
- Requires Remote Procedure Call (RPC) and port mapper services for request handling
- External Data Representation (XDR) is needed for data serialization

# NFS Architecture



# Sketch of Access Protocol

1. Remote file system is mounted on client
  - Mount program obtains a handle for remote directory from NFS server
  - Local kernel creates a new vnode for the remote directory in the VFS layer
2. File open
  - Parent portion of file's pathname translates to the vnode
  - Server address is retrieved from vnode
  - Lookup request for the remaining path portion is sent to the server
  - Open file entry is created on client using file handle and attributes supplied by the server (no file *open* is performed on the remote end)
  - File descriptor is returned to the (client) application
3. Data access
  - Local file cache is checked for data availability (if data caching is enabled)
  - If that fails, server is asked to provide the data (performs file open and read) for file handle associated with local descriptor
  - Data arrive at client and are stored in the cache
  - User buffer is filled as appropriate and call returns

Server maintains replay cache to keep track of non-idempotent requests (e.g., file delete). Matching request with a cache entry causes re-transmission of the recorded reply without re-execution of the request.

# NFS Revisions

- First public release: NFSv2 (late 1980s)
  - Outdated (32-bit offsets and file handles, poor write performance due to blocking, small payloads)
- NFSv3
  - 64-bit file offsets
  - Weak cache consistency scheme
  - Correct support for ACLs from ACL-agnostic clients
  - Larger data payload per packet
  - Still in use in some installations
- NFSv4 (current)
  - Servers no longer stateless
  - File locks respected
  - Support for lease-based byte-range locking
  - Caching of file data via delegation mechanism (multiple readers, single writer) with revocation on conflict
  - Compound RPC (request merging)
  - Kerberos 5 and SPKM/LIPKEY security
  - File migration and replication

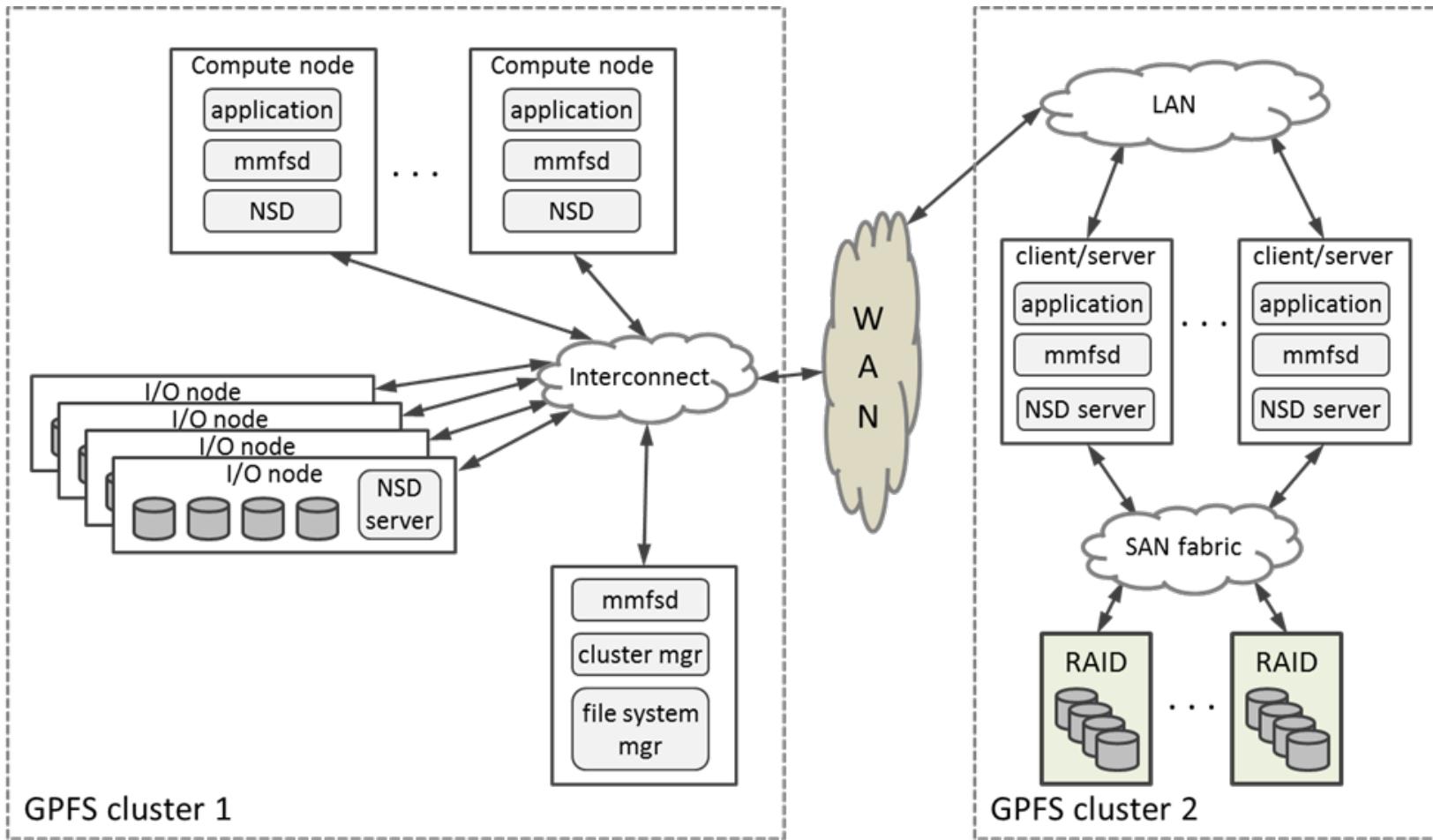
# General Parallel File System (GPFS)

- Developed by IBM and released in the late 1990s
- Influenced by Tiger Shark file system and Vesta parallel file system (both IBM products)
- Renamed in 2015 to IBM Spectrum Scale
- Available for AIX (Power architecture), Windows and Linux (x86\_64)
- Permits multiple clients to access one or multiple file system instances
- May span multiple physical devices, including remote storage over SAN or higher level protocols
- Supports parallel access to shared files as well as “shared-nothing” operation thanks to File Placement Optimizer
- High aggregate access bandwidth is accomplished by striping data across multiple storage devices and load balancing
- Supports data replication for recoverability and availability
- Multiple GPFS instances (GPFS clusters) may be configured to be visible and accessible from within a single shared namespace
- POSIX file access available via VFS integration
- Sophisticated token management for concurrent distributed locking
- Intelligent prefetching recognizing different access patterns
- Journaling for increased data integrity

# GPFS Limits

Parameter	Design limit	Tested value
Number of joined nodes per cluster	16,384	9,620
Number of disks per cluster	2,048	unknown
File size	$2^{99}$ bytes	approx. 18 Petabytes
Number of files per file system	$2^{64}$	9,000,000,000

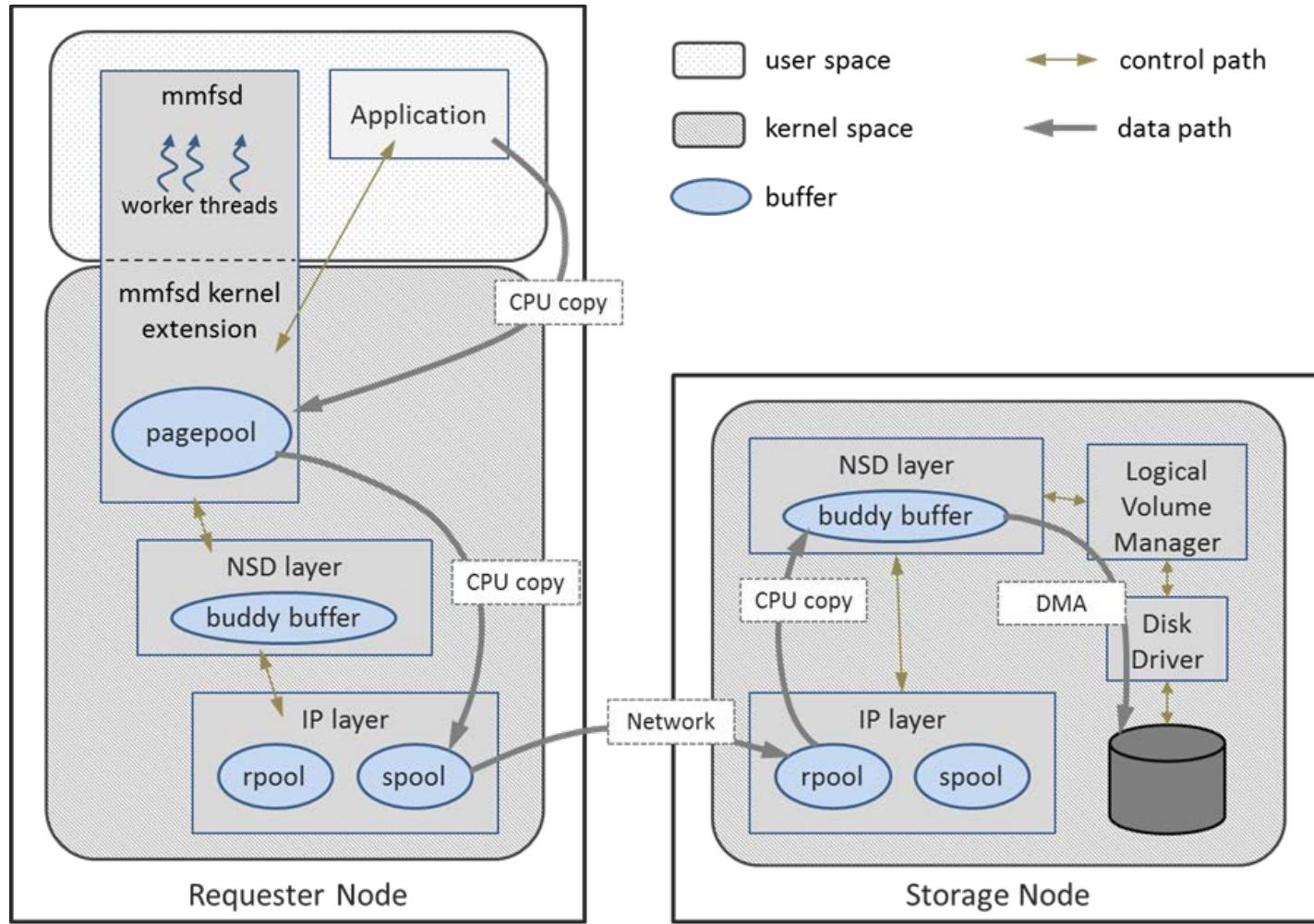
# GPFS Architecture



# Primary GPFS Components

- GPFS daemons (mmfsd)
  - Implements core functionality
  - Multithreaded with multiple thread priorities
  - May directly communicate with each other for improved synchronization
  - Responsibilities: disk space allocation for new files, directory management, lock operations for data and metadata integrity, buffer management (pagepools), I/O operation scheduling, quota accounting
- File system manager
  - One per file system, possibly distributed
  - Supervises operation of all nodes using the file system
  - Responsibilities: configuration services (storage pool expansion and repair, storage availability), storage space allocation, token management (access rights, byte-range locking), quota management
- Cluster manager
  - One per GPFS cluster
  - Elected by quorum of cluster nodes
  - Responsibilities: tracking of disk leases, monitoring node health, supervision of recovery processes, selection of file system manager nodes, propagation of configuration changes, mapping of user identities
- Network Shared Disks (NSDs)
  - Provide storage abstraction
  - Service runs on storage-equipped nodes
  - Storage components may associate with multiple servers for resilience

# File Write in GPFS



# Sketch of Write Algorithm in GPFS (I)

- Data write causes:
  - Flush command invocation
  - Synchronous write called
  - Buffers need to be reused
  - Token is revoked
  - Last byte of file block was written by sequential access
- Scenario 1: Buffer available in client memory and token is valid
  1. Data are copied from application space to available buffer (application may proceed at this point)
  2. mmfsd schedules asynchronous write to storage
  3. Buffer content is broken up into chunks fitting message payloads and copied to send pool
  4. List of destination I/O nodes is derived from metadata
  5. Data are transferred to respective NSD servers' receive pools
  6. Buddy buffer is allocated to reassemble write buffer contents
  7. Receive buffers are released
  8. Disk write is initiated

# Sketch of Write Algorithm in GPFS (II)

- Scenario 2: Buffer not available, token is valid
  1. Kernel suspends the calling thread
  2. mmfsd thread attempts to obtain buffer
    - a) If entire block is overwritten, a new buffer is used
    - b) If not, the remainder of the block is fetched into a buffer
  3. The call proceeds as in Scenario 1, steps 3-8
- Scenario 3: No buffer and no token available
  1. Acquire token for required byte range based on I/O pattern taking into consideration conflicts with other accessors
  2. Proceed as in Scenario2

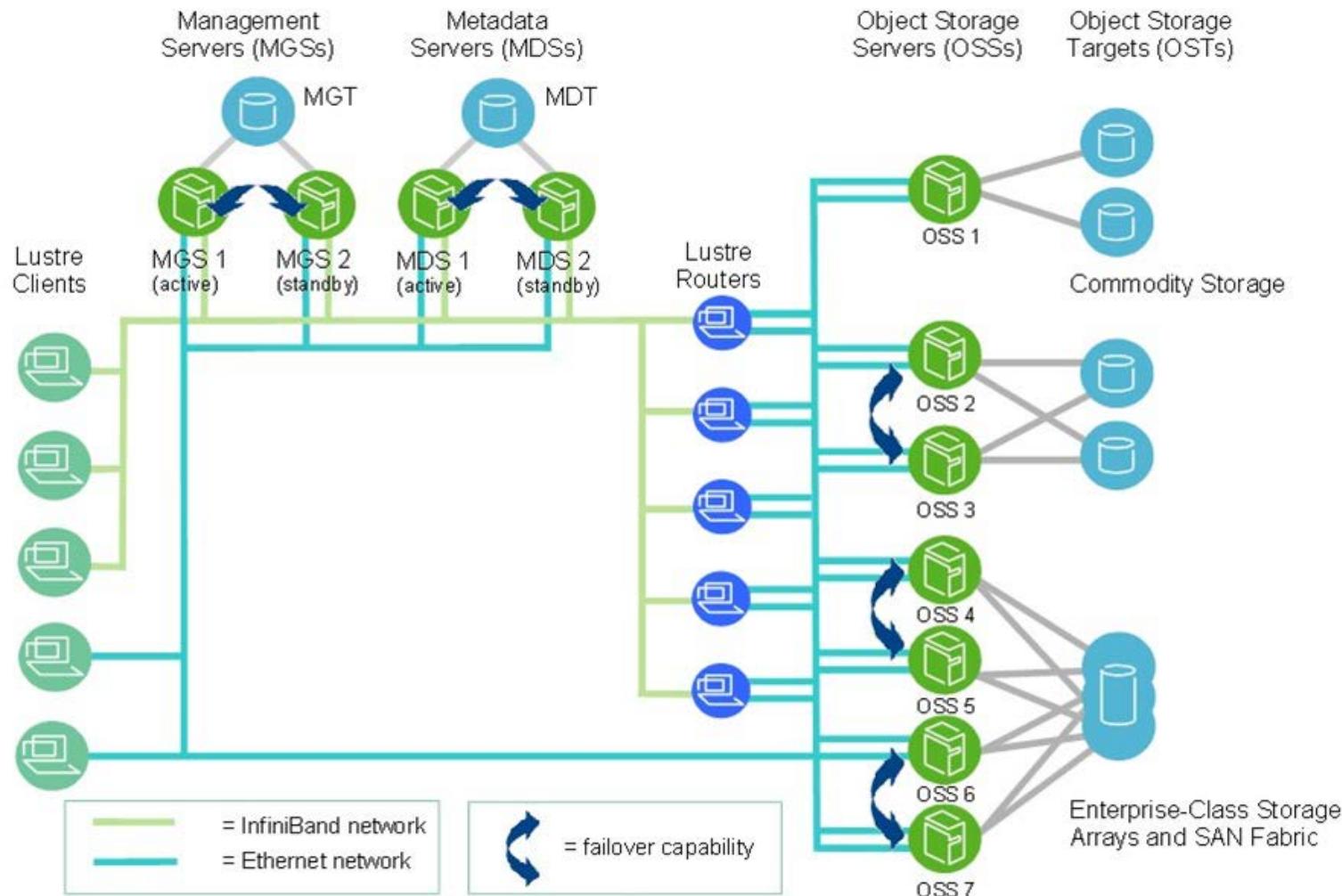
# Lustre File System

- Name derives from “Linux” and “cluster”
- First release in 2003
- Developed initially under DOE Accelerated Strategic Computing Initiative (ASCI) Path Forward
- Owned at some point in time by Sun Microsystems, Oracle, Whamcloud, and Intel
- POSIX-compliant interface, atomic semantic
- Highly scalable (tens of thousands clients, PBs of storage, I/O bandwidths in hundreds GB/s)
- Dynamic allocation of storage space and bandwidth
- Extensive network support with RDMA and bridging
- High availability with multiple failover modes and multiple-mount protection
- Transparent recovery
- Distributed online file system check
- POSIX Access Control Lists
- Fine-grain locking permitting concurrent operation on shared directories and files
- Layout control over whole file system, single directory or a single file
- Broad interoperability: specialized MPI-IO ADIO layer, NFS and CIFS exports
- Strong presence on the TOP500

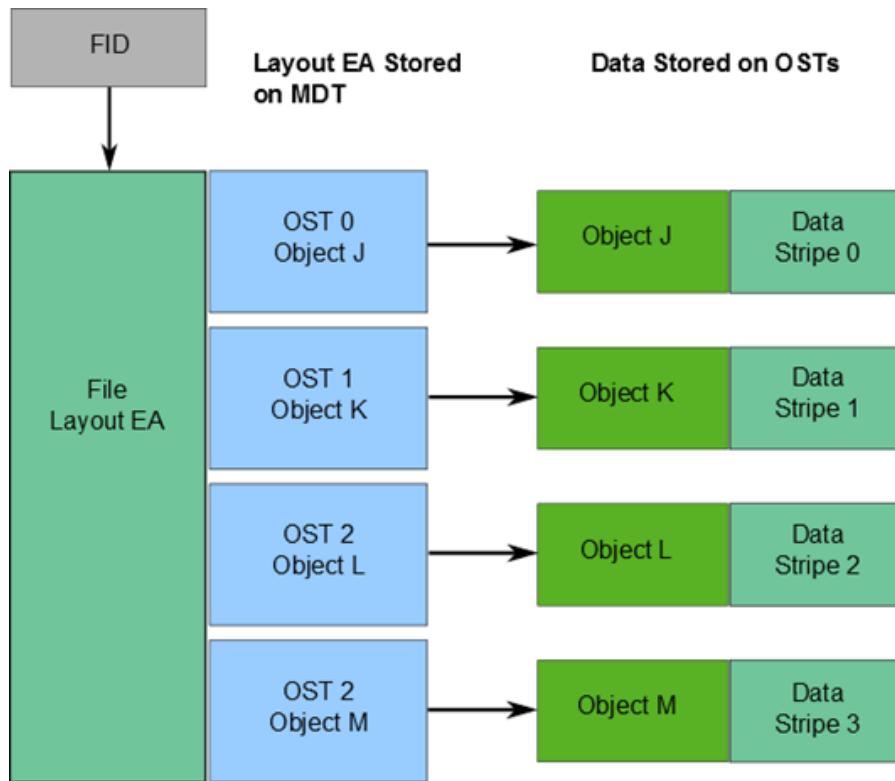
# Main System Components of Lustre

- Management Server (MGS)
  - Manages configuration information
  - Capable of sharing devices in MDS pool
- Management Target (MGT)
  - Storage space for MGS (<100MB for most installations)
  - Requires high reliability devices or device pools
- Metadata Server (MDS)
  - Name and directory content management
  - Supports distributed namespace (across multiple MDSs)
  - Provides standby MDSs as a failover option
- Metadata Target (MDT)
  - Stores directories, file names, permissions, layout information, etc.
  - Distributed operation supported, with one MDT storing the root of the file system or with striped directories across multiple MDTs
  - Typically uses a dedicated hardware node
- Object Storage Server (OSS)
  - Services data I/O requests
- Object Storage Target (OST)
  - Manages physical storage for file contents
  - 2-8 OSTs typically used per each OSS
  - Single OST capacity limited to 128TB (256TB on ZFS)
- Clients
  - Execute applications generating the I/O requests
  - Compute nodes as well as conventional desktops and workstations
- Lustre Networking (LNET)
  - Communication infrastructure
  - Supports: IB/OFED, GigE, IPoIB, Cray Seastar, Myrinet MX, Raid Array, Quadrics Elan, RDMA
  - Connectionless and asynchronous

# Example Deployment Architecture



# Lustre File Layout



- 128-bit File Identifiers (FIDs)
  - 64-bit sequence number
  - 32-bit object ID
  - 32-bit version number
- Data striped in a round-robin fashion across OSTs
- Number of stripes, stripe size, target OSTs are user configurable
- Default stripe size: 1MB
- Default stripe count: 1
- Up to 2000 objects may be defined per file

# File Access Locking

- Lustre Distributed Lock Manager (LDLM)
- Based on locking algorithm used by VAX DLM
- Operate in one of six modes:
  - Exclusive: requested by MDS before a new file is created
  - Protective Write: issued by OST to client requesting write lock
  - Protective Read: granted to client(s) that need to read or execute files
  - Concurrent Write: issued by MDS to clients who request write lock when opening file
  - Concurrent Read: used during path traversal when performing path lookups and handled by related MDS
  - Null mode
- Types of locks:
  - Extent Lock: used for OST data protection
  - Flock: supporting user-space requests for file locking
  - Inode Bit Lock: for protection of metadata attributes
  - Plain Lock: limited use

# Lustre Lock Management Overview

- Callback types:
  - Blocking callback, invoked when client requests a conflicting lock
  - Completion callback, issued when a requested lock is granted or is converted to a different mode
  - Glimpse callback, used to provide information
- Each service (OST, MDS, MGS) has its own *namespace*
- Concept of *intent* is used to enable special processing of lock operation
- Each namespace has dedicated intent handlers
- Locks are acquired through a sequence of steps:
  - 1) Check by client locking service if lock belongs to a local namespace. If so, go to (7).
  - 2) Lock enqueue RPC is sent to the appropriate server. An initial ungranted lock is created.
  - 3) Inspect if there is an intent set on lock. If not, invoke policy associated with lock function. If yes, proceed to (6).
  - 4) Check for conflicts with granted and waiting locks. If no, grant lock and invoke completion callback (the lock is acquired). Otherwise, continue.
  - 5) Invoke blocking callback for every conflicting lock. Enter lock on the waiting list with the “blocked” status.
  - 6) If lock intent is set, invoke intent handler. Return the result without further interpretation.
  - 7) Local locks are created and enqueued to check if they can be granted. If they can be granted or error is detected, return the appropriate status. Otherwise, the lock request is blocked.
- Locks in Lustre are normally held indefinitely. Their release is initiated when
  - A conflicting request is received
  - A blocking callback is issued by LDLM or
  - A blocking callback is invoked on the client node

# Lustre Design Limits

Parameter	Design target	Production tested
<b>Maximum file size</b>	31.25 PB (ldiskfs) 16 TB (32-bit ldiskfs) 8 EB (ZFS)	Multiple TB
<b>Maximum file count</b>	32 billion (ldiskfs) 256 trillion (ZFS)	2 billion
<b>Maximum storage space</b>	512 PB (ldiskfs) 1 EB (ZFS)	55 PB
<b>Number of clients</b>	≤131072	50000+
<b>Single client I/O performance</b>	90% network bandwidth	2 GB/s data I/O 1000 metadata ops/s
<b>Aggregate client I/O performance</b>	10 TB/s	2.5 TB/s
<b>OSS count</b>	1000 OSSs, up to 4000 OSTs	450 OSSs with 1000 4TB OSTs 192 OSSs with 1344 8TB OSTs 768 OSSs with 768 72TB OSTs
<b>Single OSS performance</b>	10 GB/s	6+ GB/s
<b>Aggregate OSS performance</b>	10 TB/s	2.5 TB/s
<b>MDS count</b>	≤256 MDTs, ≤256 MDSS	1 primary and 1 backup
<b>MDS performance</b>	50000 create ops/s 200000 stat ops/s	15000 create ops/s 50000 stat ops/s

# MapReduce

# MapReduce

- MapReduce is a simple programming model for enabling distributed computations including data processing on very large input datasets in a highly scalable and fault-tolerant way
- The details of the underlying parallelization in MapReduce are hidden from the programmer thereby making it easier to use
- A *map* is a functional that executes a supplied function on all members of an input list
- The results of the map function and associated groupings are passed to the *reduce* function
- The map functions, like the reduce functions, can be executed concurrently giving a significant potential for speedup.
- Distributed processing in MapReduce may be summarized in three phases: a map phase, a shuffle phase, and a reduce phase.
- The Hadoop project provides an open source implementation of the MapReduce programming model

# Map Function

- A *map* is a functional that executes a supplied function on all members of an input list.
- The map function itself returns a set of two linked data items: a key for lookup and a value.
- Example: suppose the map function counts the number of characters of an input word, returning as a key the word length and returning as a value the input word:

*this is a book about high performance computing*

Result from map function:

key	Grouped values
1	“a”
2	“is”
4	“this”, “book”, “high”
9	“computing”
11	“performance”

# Reduce Function

- The reduce function takes as an argument a key and all values associated with that key
- Like the map function, the reduce function can also be executed independently on each key and grouping of values thereby enabling embarrassingly parallel execution.
- Example: A reduction function that simply counts the number of grouped values associated with each key. Using the output from the previous map function, this gives:

key	Output from reduce function
1	1
2	1
4	3
9	1
11	1

Thus there are three words of length four while only one word of each of the other counts.

# Word Count Example

- Counting the number of times each word has been used in a body of text is the canonical didactic example for MapReduce.
- The map function returns as a key a single word and the associated value with the key is unity.
- Example: *To be or not to be – that is the question*

Key	Grouped Values
“to”	1, 1
“be”	1, 1
“or”	1
“not”	1
“that”	1
“is”	1
“the”	1
“question”	1

- The reduce function simply sums up the grouped values

Key	Output from Reduce function
“to”	2
“be”	2
“or”	1
“not”	1
“that”	1
“is”	1
“the”	1
“question”	1

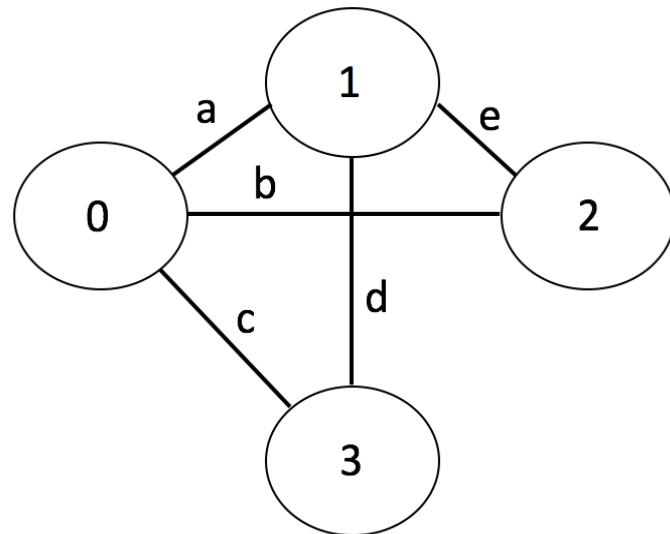
# Word Count: Hamlet

Running the Word Count map and reduce function on the entire text of Shakespeare's Hamlet gives the following word counts for some common words:

Key	Output from Reduce function
“but”	269
“as”	222
“be”	210
“England”	21
“Norway”	13

# Shared Neighbors

Finding shared neighbors in graph applications provides another good example of MapReduce functionality



In this graph vertex “0” shares a common neighbor with vertex “2”; this common neighbor is vertex “1”. MapReduce can be used to find those shared neighbors.

# Shared Neighbors: Map

- The map function returns each edge of a vertex as a key
- The value for each key is the list of all the neighboring vertices to that vertex

Vertex 0		Vertex 1		Vertex 2		Vertex 3	
key	values	key	values	key	values	key	values
a	1,2,3	a	0,2,3	e	0,1	c	0,1
b	1,2,3	d	0,2,3	b	0,1	d	0,1
c	1,2,3	e	0,2,3				

- This gives the following group values:

key	Grouped Values
a	(1,2,3), (0,2,3)
b	(1,2,3), (0,1)
c	(1,2,3), (0,1)
d	(0,2,3), (0,1)
e	(0,2,3), (0,1)

# Shared Neighbors: Reduce

- The reduce function returns the intersection of each key's grouped values revealing shared neighbors

key	Output from reduce function
a	2,3
b	1
c	1
d	0
e	0

- Here vertices connected by edge “a” also share two of the same neighbors, vertices 2 and 3.

# K-means clustering Example

Beginning with the sample data and two initially supplied cluster points (0,0) and (1,1)

Individual	(x,y) pair
a	(0.1,0.3)
b	(1.1,0.4)
c	(0.8,0.7)
d	(1.2,1.2)
e	(0.1,1.1)

Using the Euclidean distance measure,  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ , each individual is assigned to the cluster nearest to the (x,y) pair:

Cluster 1		Cluster 2	
Members	Mean x,y value	Members	Mean x,y value
a	(0.1,0.3)	b,c,d,e	(0.8,0.85)

# K-means clustering

- In a MapReduce programming model, for a given (x,y) value pair the mapper iterates over each cluster's mean value and finds the cluster with the nearest distance to the (x,y) pair. It returns as a key the cluster and as a value the (x,y) pair:

key	Grouped values
1	(0.1,0.3)
2	(1.1,0.4), (0.8,0.7), (1.2,1.2), (0.1,1.1)

- The reducer receives a list of (x,y) value pairs for each cluster and computes the new cluster mean value.

key	Output from reduce function
1	(0.1,0.3)
2	(0.8,0.85)

- This MapReduce operation can be performed iteratively until no more updates occur or until a maximum number of iterations is reached.

# Hadoop

- The Hadoop project by Apache provides an open source implementation of the MapReduce programming model.
- It provides a distributed file system, job scheduling and resource management tools including YARN (Yet Another Resource Negotiator), and MapReduce programming support.
- Historically, MapReduce applications in Hadoop are programmed using Java although support for C++, Python, and a few other languages are also available.
- The distributed file system in Hadoop, HDFS, enables distributed file access across many linked storage devices in an easy way.

# Hadoop File System Commands

Select Hadoop Distributed File System Commands	Description
<b>hdfs dfs –cat &lt;filename&gt;</b>	Copies the specified filename to stdout.
<b>hdfs dfs –ls</b>	HDFS equivalent of Linux “ls” command.
<b>hdfs dfs –mkdir &lt;directory&gt;</b>	HDFS equivalent of Linux “mkdir” command
<b>hdfs dfs –put &lt;local files&gt;... &lt;destination&gt;</b>	Copy the source files to the destination path in the HDFS
<b>hdfs dfs –get &lt;src&gt; &lt;destination&gt;</b>	Copy the source file to the local file system destination.
<b>hdfs dfs –rm &lt;filenames&gt;</b>	Delete the specified files. Only deletes files.
<b>hdfs dfs –rmdir &lt;directory name&gt;</b>	Delete the specified directory and all content.

Example: place hamlet.txt in the Hadoop distributed file system:

```
hdfs dfs -put hamlet.txt /hamlet
```

# Checkpointing

# Overview

- Applications with long execution times run a significant risk of encountering a hardware or software failure before the completion.
- Long execution times also frequently violate some supercomputer usage policies where a maximum wallclock limit for a simulation is established.
- The consequences from a hardware or software failure can be very significant and costly in terms of time lost and computing resources wasted for long running jobs.
- At designated points during the execution of an application on a supercomputer the data necessary to allow later resumption of the application at that point in the execution can be output and saved. This data is called a checkpoint.

# Uses for checkpoint files

- Checkpoint files help mitigate the risk of a hardware or software failure in a long running job.
- Checkpoint files also provide snapshots of the application at different simulation epochs, help in debugging, aid in performance monitoring and analysis, and can help improve load-balancing decisions for better distributed memory usage.

# Checkpointing Strategies

- In HPC applications, two common strategies for checkpoint/restart are employed: system-level checkpoint and application-level checkpointing.
- System-level checkpointing requires no modifications to the user code but may require loading a specific system-level library.
- System-level checkpointing strategies center on full-memory dumps and may result in very large checkpoint files.
- Application-level checkpointing requires modifications to the user code. Libraries exist to assist this process.
- Application-level checkpoint files tend to be more efficient since they only output the most relevant data needed for restart.

# System-level checkpointing

- System-level checkpointing performs the checkpoint and restart procedures via a full memory dump.
- This type of checkpointing does not require any changes to the application in order to use and the writing of the checkpoint may be triggered either by the system or the user.
- Examples with support for High Performance Computing include the Berkeley Lab Checkpoint/Restart (BLCR), Checkpoing/Restore in Userspace (CRIU), and Distributed MultiThreaded CheckPointing (DMTCP).

# Example: System-level checkpointing with DMTCP

```
1 #include <omp.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <math.h>
6
7 int main (int argc, char *argv[])
8 {
9     const int size = 200;
10    int nthreads, threadid, i;
11    double array1[size], array2[size], array3[size];
12
13    // Initialize
14    for (i=0; i < size; i++) {
15        array1[i] = 1.0*i;
16        array2[i] = 2.0*i;
17    }
18
19    int chunk = 3;
20
21 #pragma omp parallel private(threadid)
22 {
23     threadid = omp_get_thread_num();
24     if (threadid == 0) {
25         nthreads = omp_get_num_threads();
26         printf("Number of threads = %d\n", nthreads);
27     }
28     printf(" My threadid %d\n",threadid);
29
30 #pragma omp for schedule(static,chunk)
31 for (i=0; i<size; i++) {
32     array3[i] = sin(array1[i] + array2[i]);
33     printf(" Thread id: %d working on index %d\n",threadid,i);
34     sleep(1);
35 }
36
37 } // join
38
39 printf(" TEST array3[199] = %g\n",array3[199]);
40
41 return 0;
42 }
```

The code is compiled as normal:

```
gcc -fopenmp -O3 -o checkpoint_openmp checkpoint_openmp.c -lm
```

The code number of OpenMP threads is also specified in the normal way

```
export OMP_NUM_THREADS=16
```

The executable is launched with checkpoint capability using the dmtcp\_launch tool:

```
dmtcp_launch ./checkpoint_openmp
```

# Example: System-level checkpointing with DMTCP

Request for checkpoint is manually executed in the *dmtcp\_coordinator* command interface or pre-supplied as follows:

```
dmtcp_command --interval <checkpoint interval in seconds>
```

DMTCP checkpoint files have the naming convention of

ckpt\_<executable name>\_<client identity>.dmtcp

and are written in the directory where the executable was launched.

The checkpoint file is restarted using the *dmtcp\_restart* command:

```
dmtcp_restart <checkpoint file>
```

# Example: System-level checkpointing with DMTCP

```
andersmw@cutter:~/textbook$ dmtcp_restart ckpt_checkpoint_openmp_16707112e4c8f-42000-8687a700c18a5.dmtcp
Thread id: 15 working on index 141
Thread id: 14 working on index 138
Thread id: 7 working on index 117
Thread id: 12 working on index 132
Thread id: 1 working on index 99
Thread id: 2 working on index 102
Thread id: 8 working on index 120
Thread id: 5 working on index 111
Thread id: 11 working on index 129
Thread id: 13 working on index 135
Thread id: 3 working on index 105
Thread id: 10 working on index 126
Thread id: 9 working on index 123
Thread id: 6 working on index 114
Thread id: 4 working on index 108
Thread id: 0 working on index 96
                                         Thread id: 15 working on index 141
                                         Thread id: 14 working on index 138
                                         Thread id: 8 working on index 120
                                         Thread id: 1 working on index 99
                                         Thread id: 2 working on index 102
                                         Thread id: 4 working on index 108
                                         Thread id: 12 working on index 132
                                         Thread id: 7 working on index 117
                                         Thread id: 0 working on index 96
                                         Thread id: 13 working on index 135
                                         Thread id: 11 working on index 129
                                         Thread id: 5 working on index 111
                                         Thread id: 3 working on index 105
                                         Thread id: 9 working on index 123
                                         Thread id: 6 working on index 114
                                         Thread id: 10 working on index 126
```

The standard output from the example code after checkpoint restart (left) and without restart (right). The same OpenMP threads operate on the same indices and all operations are identical between the restarted and non-restarted case.

# Application level checkpointing

- In application-level checkpointing the application developer has the responsibility to perform all checkpoint/restart operations.
- While inconvenient, application-level checkpoint/restart tends to produce checkpoint files that are smaller than system-level checkpoint/restart where a full memory dump is performed.
- For distributed memory applications based on MPI, application-level checkpoint/restart approaches often share some basic characteristics:
  - Only one checkpoint file is written per MPI process
  - Only one MPI rank accesses a single checkpoint file
  - Checkpoint files do not contain data from multiple checkpoint epochs
  - Checkpoint files are generally written to the parallel file system by the compute nodes
  - Checkpoint/restart overheads can be large.

# Application level checkpointing

- Application-level checkpoint/restart implementations generally pick designated points in the computational phase in the simulation algorithm for checkpointing in order to ensure computational phase consistency in the checkpoint epoch
- Application-level checkpoint/restart is very popular in large-scale MPI applications and toolkits because it can be tailored for the application to be as efficient and minimal as possible.
- The Scalable Checkpoint/Restart (SCR) library assists application-level checkpoint strategies by reducing the load on a parallel file system and by partially utilizing non-parallel fast storage local to a compute node for checkpoint file storage with some redundancy in the event of a failure on the local storage.

# Next Steps and Beyond

# Looking towards Exascale Computing

- Projected for 2023 (US); maybe earlier for China
- Exascale Hardware Systems
  - Achieving  $10^{18}$  flops sustained
  - Billion-way parallelism required
  - Requires dramatic improvements in power efficiency – 20 Megawatts
  - Retaining resiliency in the presence of faults
- Scalable Parallel Programming
  - X-10
  - Chapel
  - XPI
  - Hybrid programming: MPI + X
- Runtime Software Systems
  - Exploits runtime information and dynamic control for efficiency and scalability
  - ParalleX/HPX
  - UIUC – Charm++
  - Rice University - Open Community Runtime (OCR)

# Beyond Exascale

- Quantum Computing
  - Exploits quantum mechanics
  - Uses superposition, tunneling, entanglement
  - Might solve problems otherwise impossible
  - Still in research, but promising
- Neuromorphic Computing
  - Brain inspired
  - Neural networks early example
  - Uses training sets in many cases
- Superconducting Supercomputing
  - Josephson junctions for quantum flux gates
  - Potentially 100's of GigaHertz clock rate
- Processor in Memory (PIM)
  - Merges logic at sense amps of memory
  - Reduces latency, increases bandwidths

