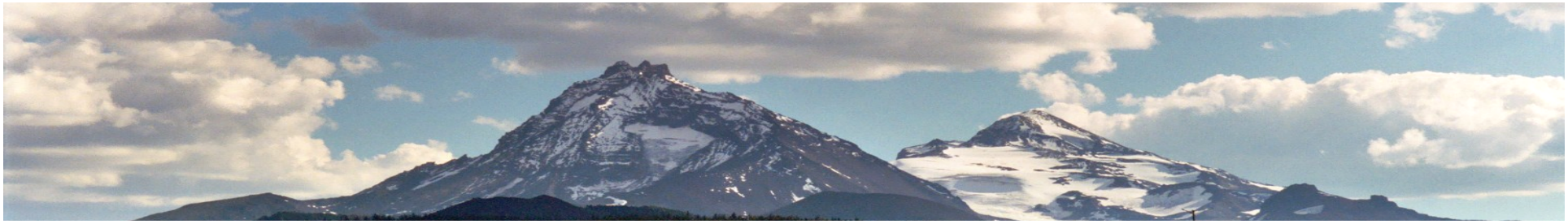


Lecture 1:
Introduction and Fundamentals

Allen D. Malony

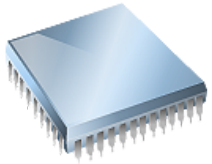
Department of Computer and Information Science

UNIVERSITY OF OREGON

What is Performance?

- In computing, performance is defined by 2 factors
  - Computational requirements (what needs to be done)
  - Computing resources (what it costs to do it)
- Computational problems translate to requirements
- Computing resources interplay and tradeoff

$$Performance \sim \frac{1}{Resources\ for\ solution}$$



Hardware

Time

Energy

… and ultimately

Money

Why do we care about Performance?

- Performance itself is a measure of how well the computational requirements can be satisfied
- We evaluate performance to understand the relationships between requirements and resources
  - Decide how to change "solutions" to target objectives
- Performance measures reflect decisions about how and how well "solutions" are able to satisfy the computational requirements

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."
Charles Babbage, 1791 – 1871

What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
  - Performance with respect to parallel computation
- Performance is the *raison d'être* for parallelism
  - Parallel performance versus sequential performance
  - If the "performance" is not better, parallelism is not necessary
- *Parallel processing* includes techniques and technologies necessary to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …
- Parallelism must deliver performance
  - How? How well?

Concurrency and Parallelism

- Concurrent execution
  - Multiple tasks *can* execute at the same time
  - There are no dependencies between the tasks
- Parallel execution
  - Concurrent tasks *actually* execute at the same time
  - Multiple processing resources are available
- Find concurrent execution opportunities
- Develop support for enabling parallel execution
- Parallelism granularity
  - Processes, threads, routines, statements, instructions, …

Why use parallel processing?

- Two primary reasons (both performance related)
  - Faster time to solution (response time)
  - Solve bigger computing problems in same time
- Other factors motivate parallel processing
  - Effective use of machine resources
  - Cost efficiencies
  - Overcoming memory constraints
- Serial machines have inherent limitations
  - Processor speed
  - Memory bottlenecks
- Parallelism has become the future of computing
- Performance is still the driving concern

Performance Expectations

- If each processor is rated at k MFLOPS and there are p processors, we should expect to see k*p MFLOPS performance? Correct?
- If it takes 100 seconds on 1 processor, it should take 10 seconds on 10 processors? Correct?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - solution to one problem may create another
    - one problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand *performance space*

Scalability

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?
- Apply performance metrics

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system
- $S(p)$ (S_p) is the *speedup*

$$S(p) = \frac{T_1}{T_p}$$

- $E(p)$ (E_p) is the *efficiency*

$$Efficiency = \frac{S_p}{p}$$

- $Cost(p)$ (C_p) is the *cost*

$$Cost = p \times T_p$$

- Parallel algorithm is *cost-optimal*
  - *Parallel time = sequential time* ($C_p = T_1$, $E_p = 100\%$)

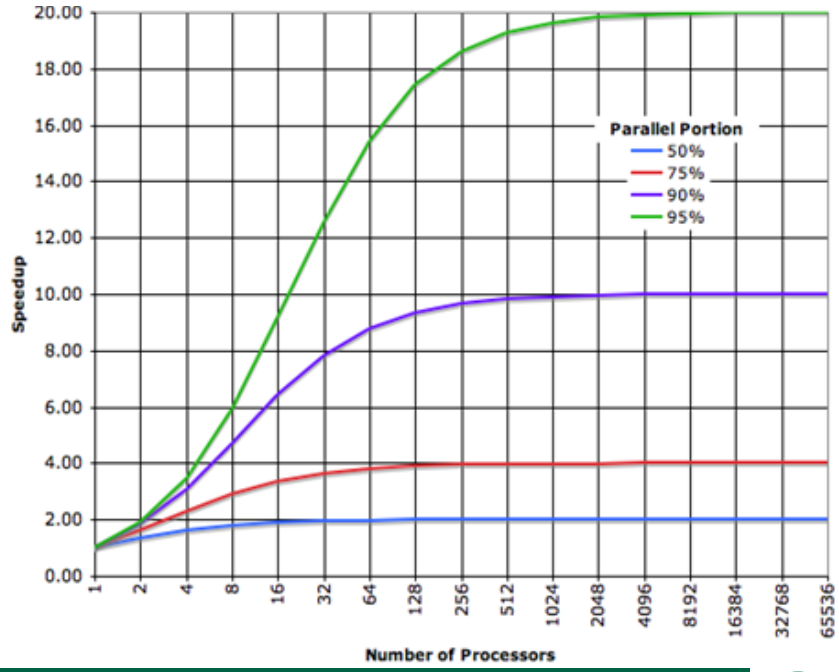
Amdahl's Law (Fixed Size Speedup)

- Let f be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let T_1 be the execution time on 1 processor
- Let T_p be the execution time on p processors
- S_p is the *speedup*

$$S_p = T_1 / T_p$$
$$= T_1 / (fT_1 + (1-f)T_1/p))$$
$$= 1 / (f + (1-f)/p))$$

- As $p \rightarrow \infty$

$$S_p = 1/f$$



Performance and Scalability

- Evaluation
  - Sequential runtime (T_{seq}) is a function of
    - problem size and architecture
  - Parallel runtime (T_{par}) is a function of
    - problem size and parallel architecture
    - # processors
  - Parallel performance affected by
    - algorithm + architecture
- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional with respect to the number of processors
- Amdahl's Law applies when the problem size is fixed
  - *Strong scaling* ($p \rightarrow \infty$, $S_p = S_\infty \rightarrow 1 / f$)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!

Gustafson's Law (Scaled Speedup)

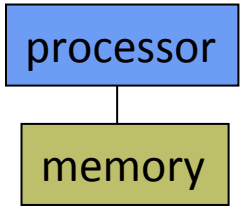
- Often interested in running larger problems when scaling
- Problem size is determined by constraint on parallel time
- Assume parallel time is kept constant
  - $T_p = C = (f + (1-f)) * C$
  - f_{seq} is the fraction of T_p spent in sequential execution
  - f_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
  - Let $C=1$, then $T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p\text{-}1)f_{par}$
- What is the speedup in this case?
  - $S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p\text{-}1)f_{par}$
- Gustafson's Law applies when the problem size can increase as the number of processors increase
  - *Weak scaling* ($S_p = 1 + (p\text{-}1)f_{par}$)

Scalable Parallel Computing

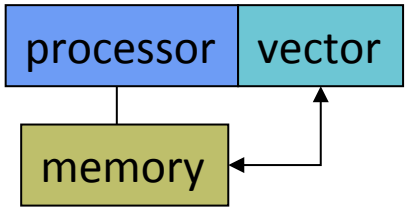
- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks
- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - Computation to memory access ratio
    - Computation to communication ratio
- Parallel programming models and tools
- Performance scalability

Parallel Architecture Types – 1

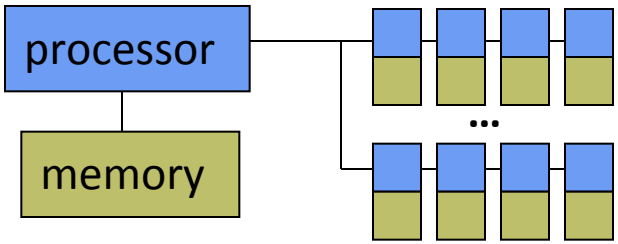
- Uniprocessor
  - Scalar processor



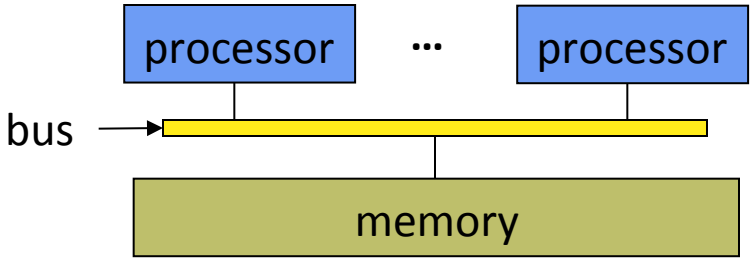
  - Vector processor



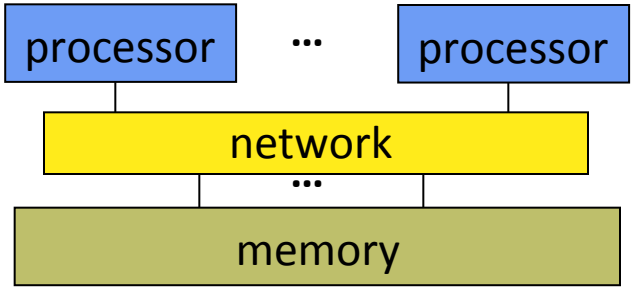
  - Single Instruction Multiple Data (SIMD)



- Shared Memory Multiprocessor (SMP)
  - Shared address space
  - Bus-based memory system

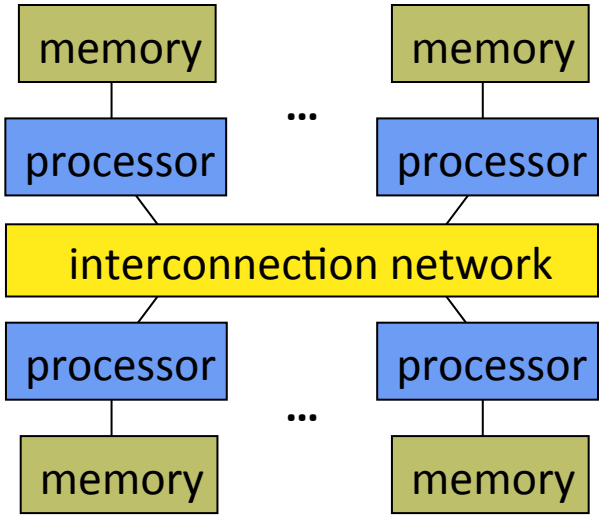


  - Interconnection network



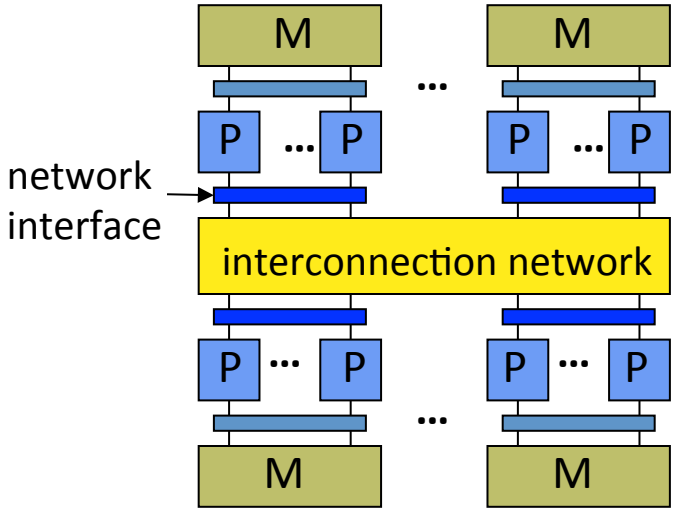
Parallel Architecture Types – 2

- Distributed Memory Multiprocessor (DMP)
  - Message passing between nodes



  - Massively Parallel Processor (MPP)

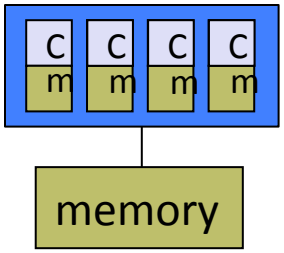
- Clusters of SMPs
  - Shared address space on SMP node
  - Message passing between nodes



  - Can also be a MPP

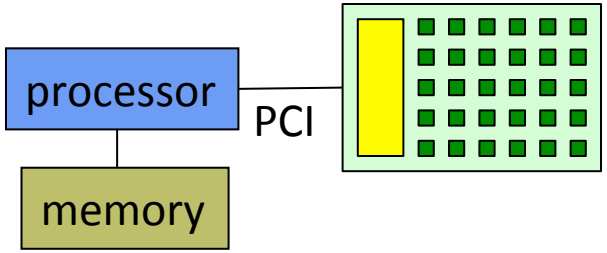
Parallel Architecture Types – 3

- Multicore (Manycore)
  - Multicore processor

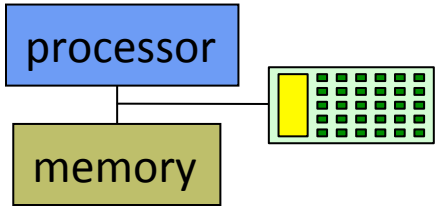


cores can be hardware multithreaded (hyperthread)

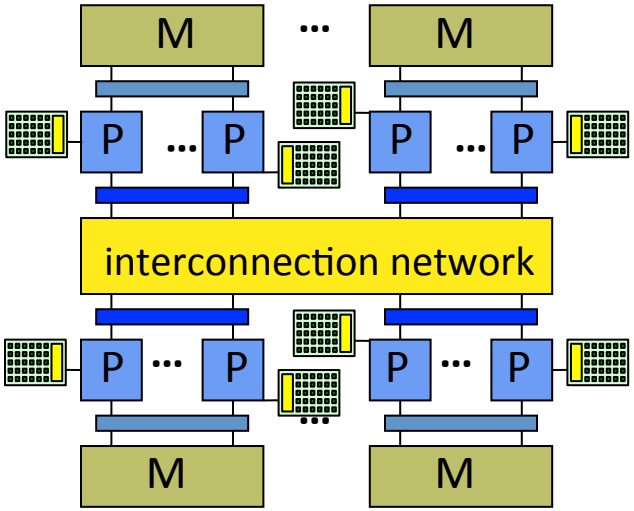
  - Accelerator / coprocessor



  - "Fused" processor



- Multicore Heterogeneous Cluster
  - Shared address space on SMP node and message passing between nodes
  - CPUs + accelerator nodes



How do you get parallelism in hardware?

- Instruction-Level Parallelism (ILP)
- Data parallelism
  - Increase amount of data to be operated on at same time
- Processor parallelism
  - Increase number of processors
- Memory system parallelism
  - Increase number of memory units
  - Increase bandwidth to memory
- Communication parallelism
  - Increase amount of interconnection between elements
  - Increase communication bandwidth

Embarrassingly Parallel Computations

- An *embarrassingly parallel* computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes *T* time sequentially, there is the potential to achieve *T/P* time running in parallel with *P* processors
  - What would cause this not to be the case always?

Why Are Parallel Applications not Scalable?

- Critical Paths
  - Dependencies between computations spread across processors
- Bottlenecks
  - One processor holds things up
- Algorithmic overhead
  - Some things just take more effort to do in parallel
- Communication overhead
  - Spending increasing proportion of time on communication
- Load Imbalance
  - Makes all processor wait for the "slowest" one
  - Dynamic behavior
- Speculative loss
  - Do A and B in parallel, but B is ultimately not needed

Critical Paths

- Long chain of dependence
  - Main limitation on performance
  - Resistance to performance improvement
- Diagnostic
  - Performance stagnates to a (relatively) fixed value
  - Critical path analysis
- Solution
  - Eliminate long chains if possible
  - Shorten chains by removing work from critical path

Bottlenecks

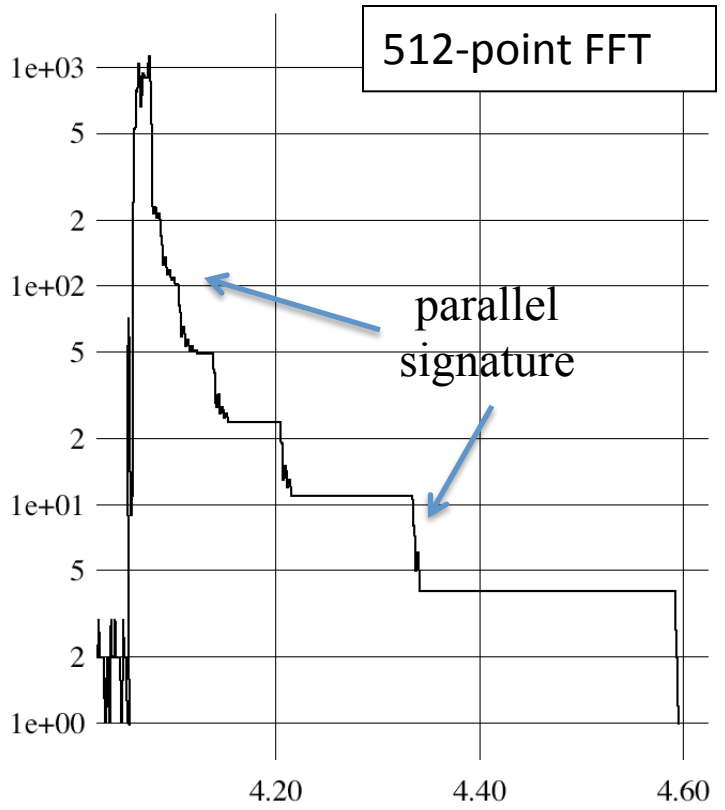
- How to detect?
  - One processor A is busy while others wait
  - Data dependency on the result produced by A
- Typical situations:
  - N-to-1 reduction / computation / 1-to-N broadcast
  - One processor assigning job in response to requests
- Solution techniques:
  - More efficient communication
  - Hierarchical schemes for master slave
- Program may not show ill effects for a long time
- Shows up when scaling

Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce addition operations (Why?)
  - *Parallel overhead*
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it doesn't parallelize well (e.g., not scalable)
- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms
- Performance is the rub
  - Are you achieving better parallel performance?
  - Must compare with the best sequential algorithm

What is the maximum parallelism possible?

- Depends on application, algorithm, program
  - Data dependencies in execution
- MaxPar analyzes the earliest possible "time" any data can be computed
  - Assumes a time model
  - Result is the maximum parallelism available
- Parallelism varies!



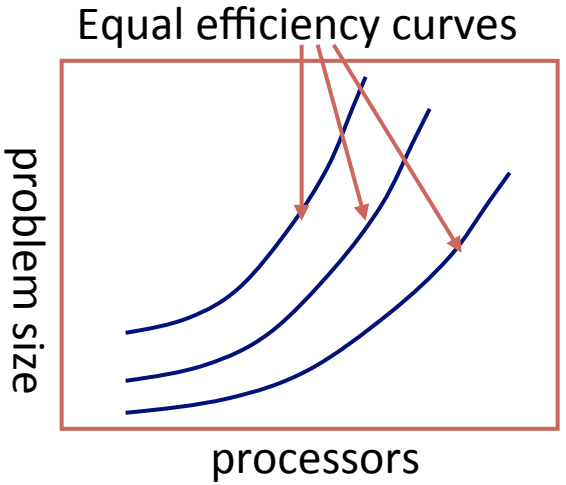
Ding-Kai Chen, "MaxPar: An Execution-driven Simulator for Studying," Master's thesis dissertation, University of Illinois, Urbana-Champaign, 1989.

Analytical / Theoretical Techniques

- Involves simple algebraic formulas and ratios
  - Typical variables are:
    - data size (N), number of processors (P), machine constants
  - Model performance of individual operations, components, algorithms in terms of the above
    - be careful to characterize variations across processors
    - model them with max operators
  - Constants are important in practice
    - Use asymptotic analysis carefully
- Scalability analysis
  - Isoefficiency (Kumar)

Isoefficiency

- How to quantify scalability
- How much increase in problem size is needed to retain the same efficiency on a larger machine?
- Efficiency
  - $T_1 / (p * T_p)$
  - T_p = computation + communication + idle
- Isoefficiency
  - Equation for equal-efficiency curves
  - If no solution
    - problem is not scalable in the sense defined by isoefficiency



Problem Size and Overhead

- Informally, problem size is expressed as a parameter of the input size
- A consistent definition of the size of the problem is the total number of basic operations (T_{seq})
  - Also refer to problem size as "work ($W = T_{seq}$)
- Overhead of a parallel system is defined as the part of the cost not in the best serial algorithm
- Denoted by T_O, it is a function of W and p

$$T_O(W,p) = pT_{par} - W \qquad (pT_{par} \text{ includes overhead})$$

$$T_O(W,p) + W = pT_{par}$$

Isoefficiency Function

- With a fixed efficiency, W is as a function of p

$$T_{par} = \frac{W + T_O(W,p)}{p}$$

$W = T_{seq}$

$$Speedup = \frac{W}{T_{par}} = \frac{Wp}{W + T_O(W,p)}$$

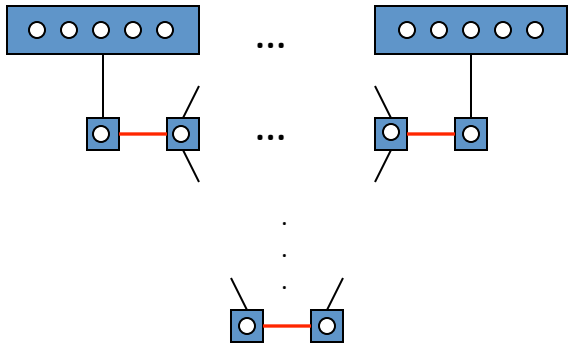
$$Efficiency = \frac{S}{p} = \frac{W}{W + T_O(W,p)} = \frac{1}{1 + \frac{T_O(W,p)}{W}}$$

$$E = \frac{1}{1 + \frac{T_O(W,p)}{W}} \rightarrow \frac{T_O(W,p)}{W} = \frac{1-E}{E}$$

$$W = \frac{E}{1-E} T_O(W,p) = K T_O(W,p) \quad \text{Isoefficiency Function}$$

Isoefficiency Function of Adding n Numbers

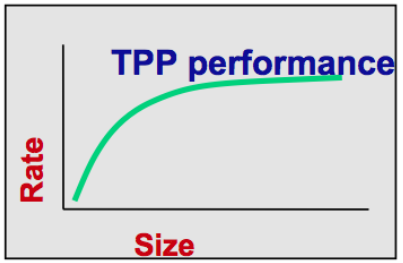
- Overhead function:
  - $T_O(W,p) = pT_{par} - W = 2plog(p)$
- Isoefficiency function:
  - $W = K*2plog(p)$



- If p doubles, W needs also to be doubled to roughly maintain the same efficiency
- Isoefficiency functions can be more difficult to express for more complex algorithms

Top 500 Benchmarking Methodology

- Listing of the world's 500 most powerful computers
- Yardstick for high-performance computing (HPC)
  - R_{max} : maximal performance Linpack benchmark
    - dense linear system of equations ($Ax = b$)
- Data listed
  - R_{peak} : theoretical peak performance
  - N_{max} : problem size needed to achieve R_{max}
  - $N_{1/2}$: problem size needed to achieve 1/2 of R_{max}
  - Manufacturer and computer type
  - Installation site, location, and year
- Updated twice a year at SC and ISC conferences

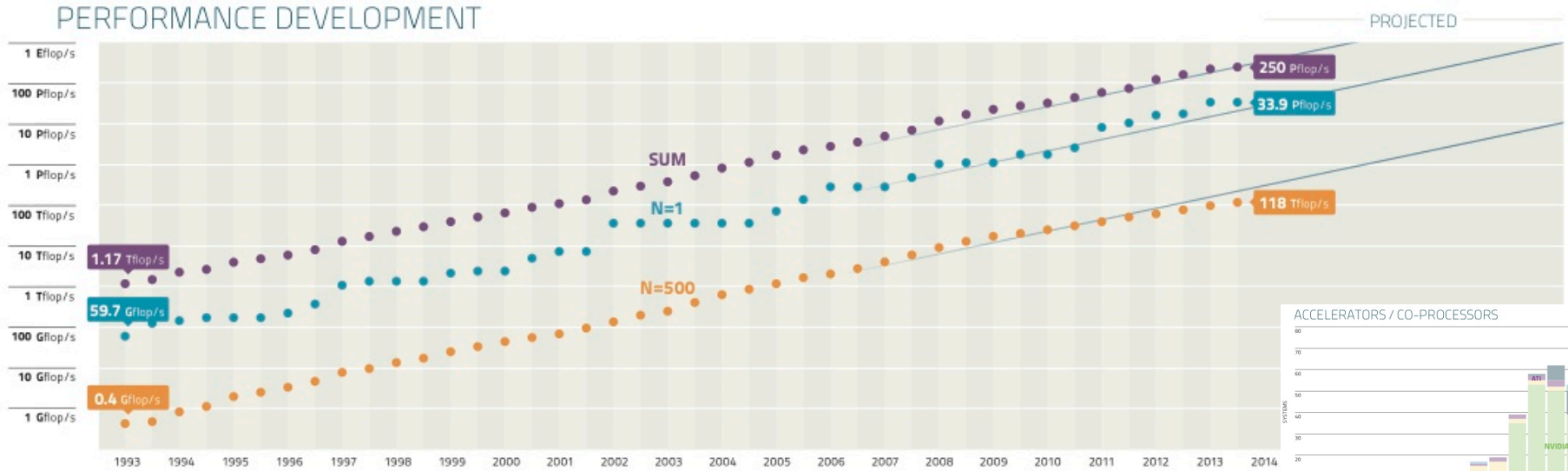


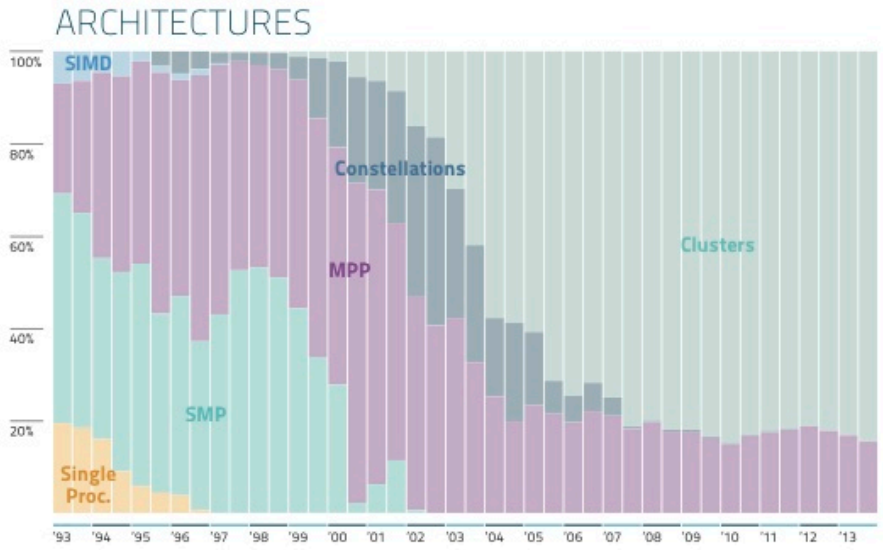
Top 10 (November 2013)

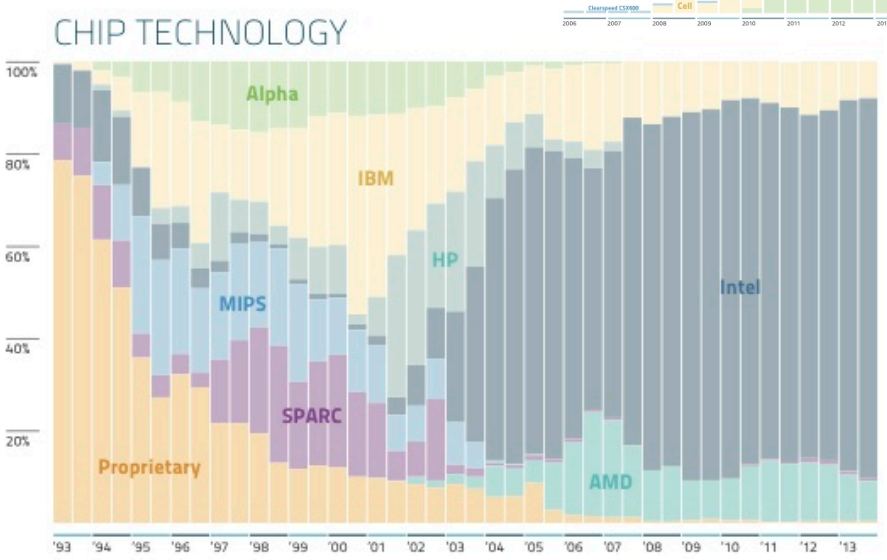
Different architectures

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972
10	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147,456	2,897.0	3,185.1	3,423

Top 500 – Performance (November 2013)



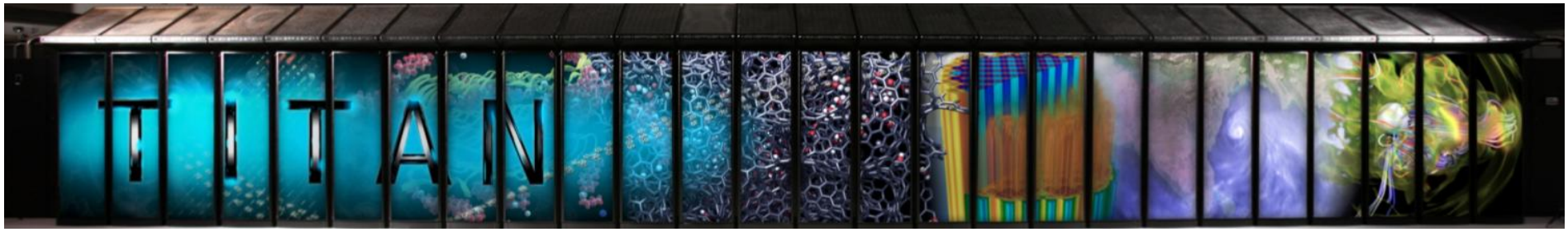




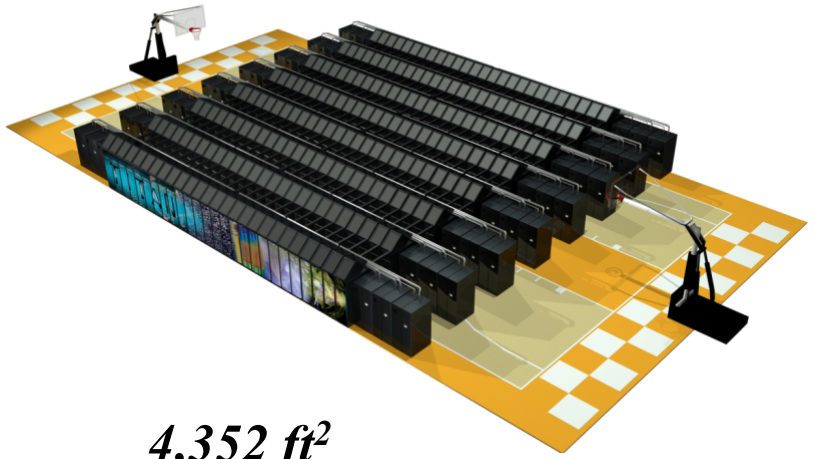
#1: NUDT Tiahne-2 (Milkyway-2)

- Compute Nodes have 3.432 Tflop/s per node
  - 16,000 nodes
  - 32000 Intel Xeon CPU
  - 48000 Intel Xeon Phi
- Operations Nodes
  - 4096 FT CPUs
- Proprietary interconnect
  - TH2 express
- 1PB memory
  - Host memory only
- Global shared parallel storage is 12.4 PB
- Cabinets: 125+13+24 =162
  - Compute, communication, storage
  - ~750 m2

#2: ORNL Titan Hybrid System (Cray XK7)



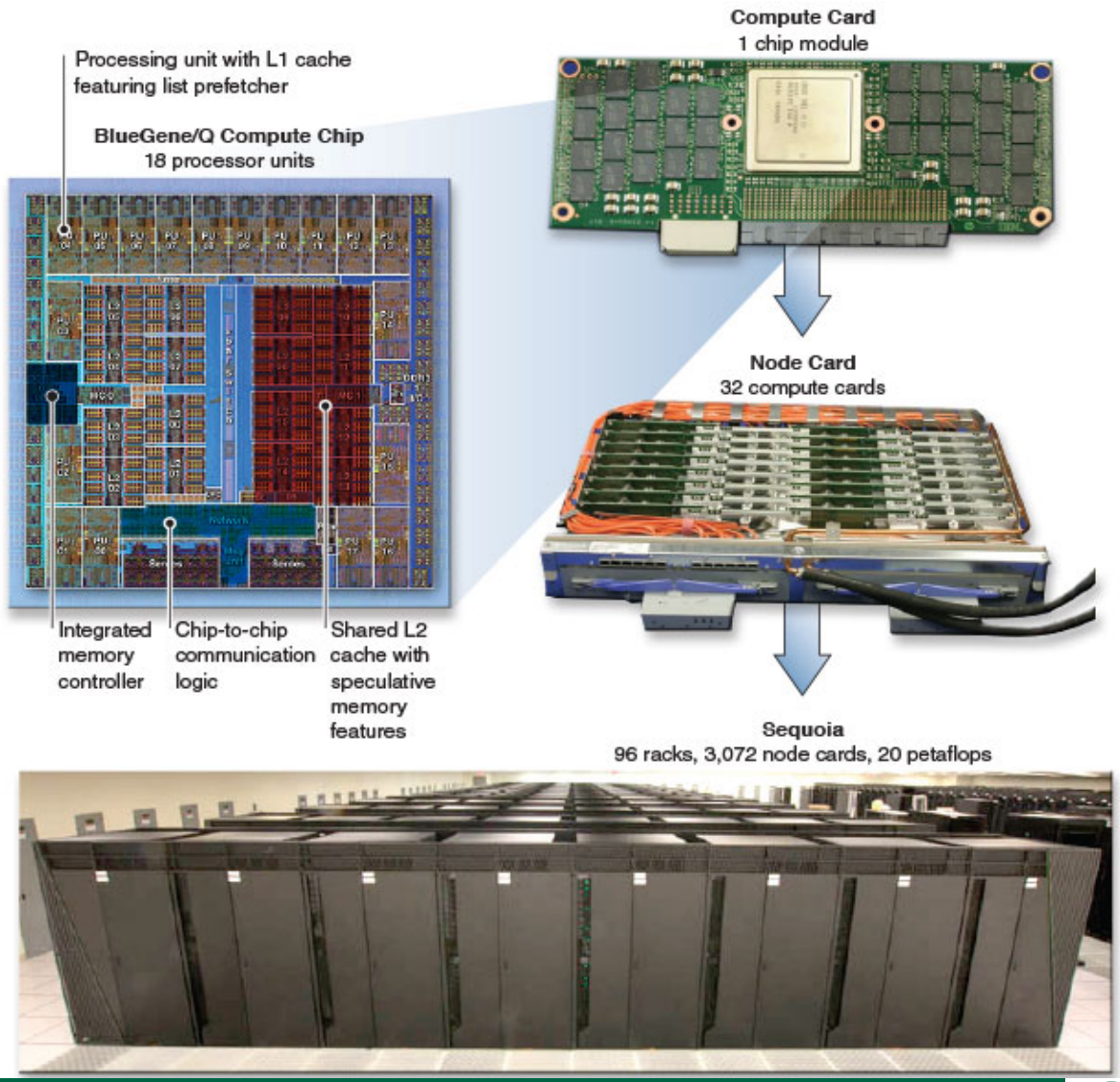
- Peak performance of 27.1 PF
  - 24.5 GPU + 2.6 CPU
- 18,688 Compute Nodes each with:
  - 16-Core AMD Opteron CPU
  - NVIDIA Tesla "K20x" GPU
  - 32 + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 8.9 MW peak power



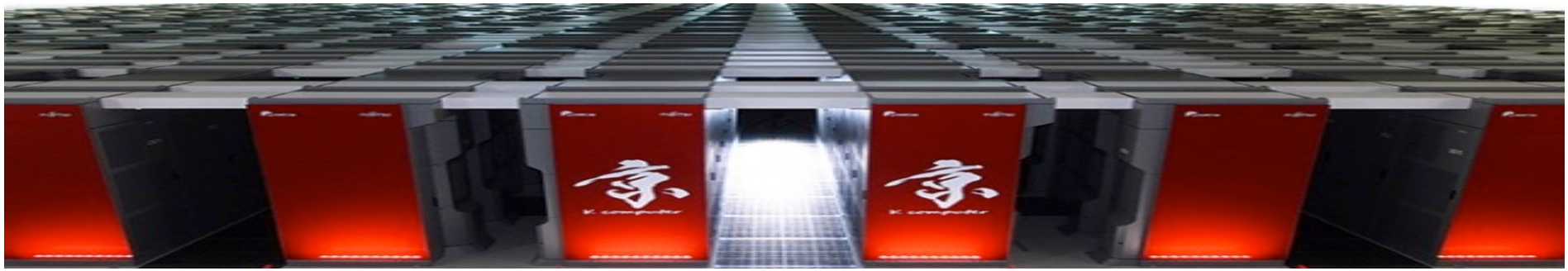
4,352 ft²

#3: LLNL Sequoia (IBM BG/Q)

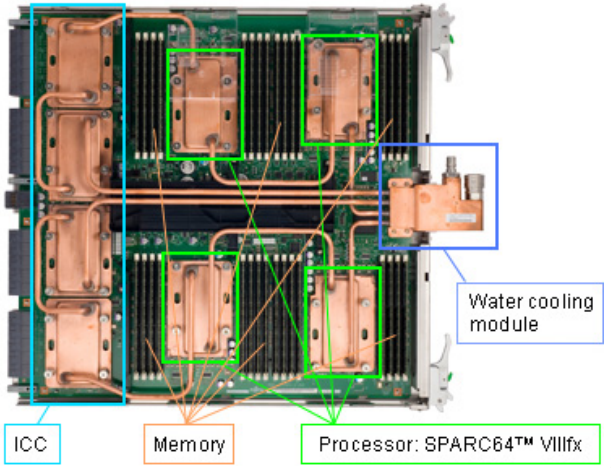
- Compute card
  - 16-core PowerPC A2 processor
  - 16 GB DDR3
- Compute node has 98,304 cards
- Total system size:
  - 1,572,864 processing cores
  - 1.5 PB memory
- 5-dimensional torus interconnection network
- Area of 3,000 ft^2

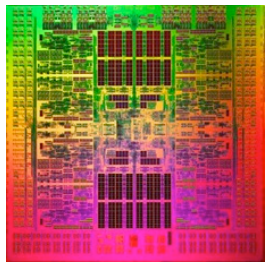


#4: RIKEN K Computer



- 80,000 CPUs
  - SPARC64 VIIIfx
  - 640,000 cores
- 800 water-cooled racks
- 5D mesh/torus interconnect (Tofu)
  - 12 links between node
  - 12x higher scalability than 3D torus





Contemporary HPC Architectures

Date	System	Location	Comp	Comm	Peak (PF)	Power (MW)
2009	Jaguar; Cray XT5	ORNL	AMD 6c	Seastar2	2.3	7.0
2010	Tianhe-1A	NSC Tianjin	Intel + NVIDIA	Proprietary	4.7	4.0
2010	Nebulae	NSCS Shenzhen	Intel + NVIDIA	IB	2.9	2.6
2010	Tsubame 2	TiTech	Intel + NVIDIA	IB	2.4	1.4
2011	K Computer	RIKEN/Kobe	SPARC64 VIIIfx	Tofu	10.5	12.7
2012	Titan; Cray XK6	ORNL	AMD + NVIDIA	Gemini	27	9
2012	Mira; BlueGeneQ	ANL	SoC	Proprietary	10	3.9
2012	Sequoia; BlueGeneQ	LLNL	SoC	Proprietary	20	7.9
2012	Blue Waters; Cray	NCSA/UIUC	AMD + (partial) NVIDIA	Gemini	11.6	
2013	Stampede	TACC	Intel + MIC	IB	9.5	5
2013	Tianhe-2	NSCC-GZ (Guangzhou)	Intel + MIC	Proprietary	54	~20

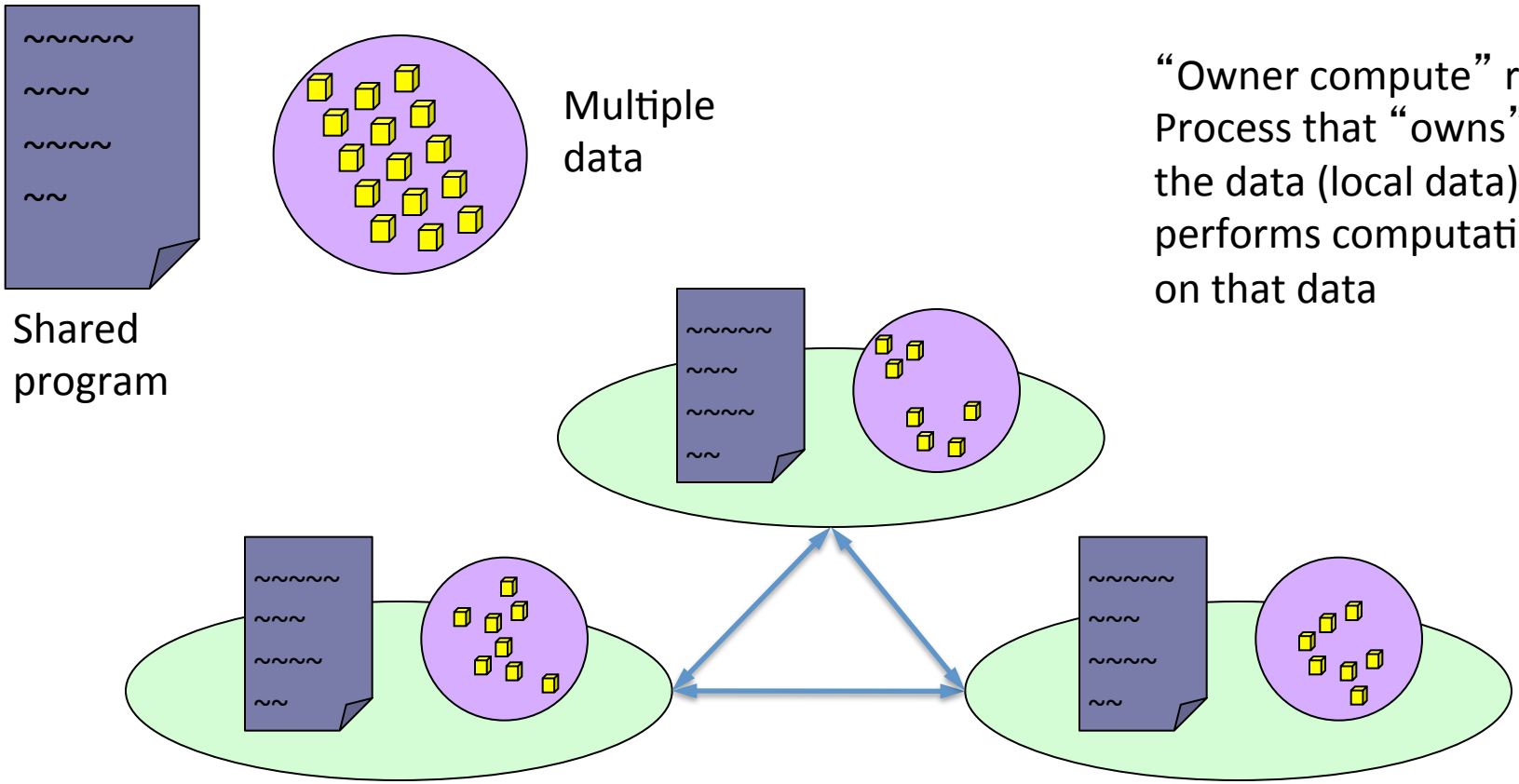
Parallel Programming Landscape

- Evolution of parallel machines has driven the advances of how parallel machines are programmed
- Parallel programming model
  - Shared memory multithreading (pthread, OpenMP, TBB)
  - Distributed memory message passing
  - Accelerator / coprocessor data parallel (CUDA, OpenACC)
  - Heterogeneous systems require a combination of methods
- Parallel software "stack"
  - Languages (must be supported by a compiler)
  - Libraries (numerical, threading, messaging, I/O, …)
  - Runtime systems (scheduling, memory management, …)
  - OS
- Parallel programming tools

Types of Parallel Computing Models

- Data parallel
  - Simultaneous execution on multiple data (SIMD)
  - Multi(many)-threaded execution on multiple data
- Task parallel
  - Different instructions on different data (MIMD)
  - Different programs on different data (MPMD)
- SPMD (Single Program, Multiple Data)
  - Combination of data parallel and task parallel
  - Not synchronized at individual operation level
- Hybrid parallel computing combines methods

SPMD (Single Program, Multiple Data)



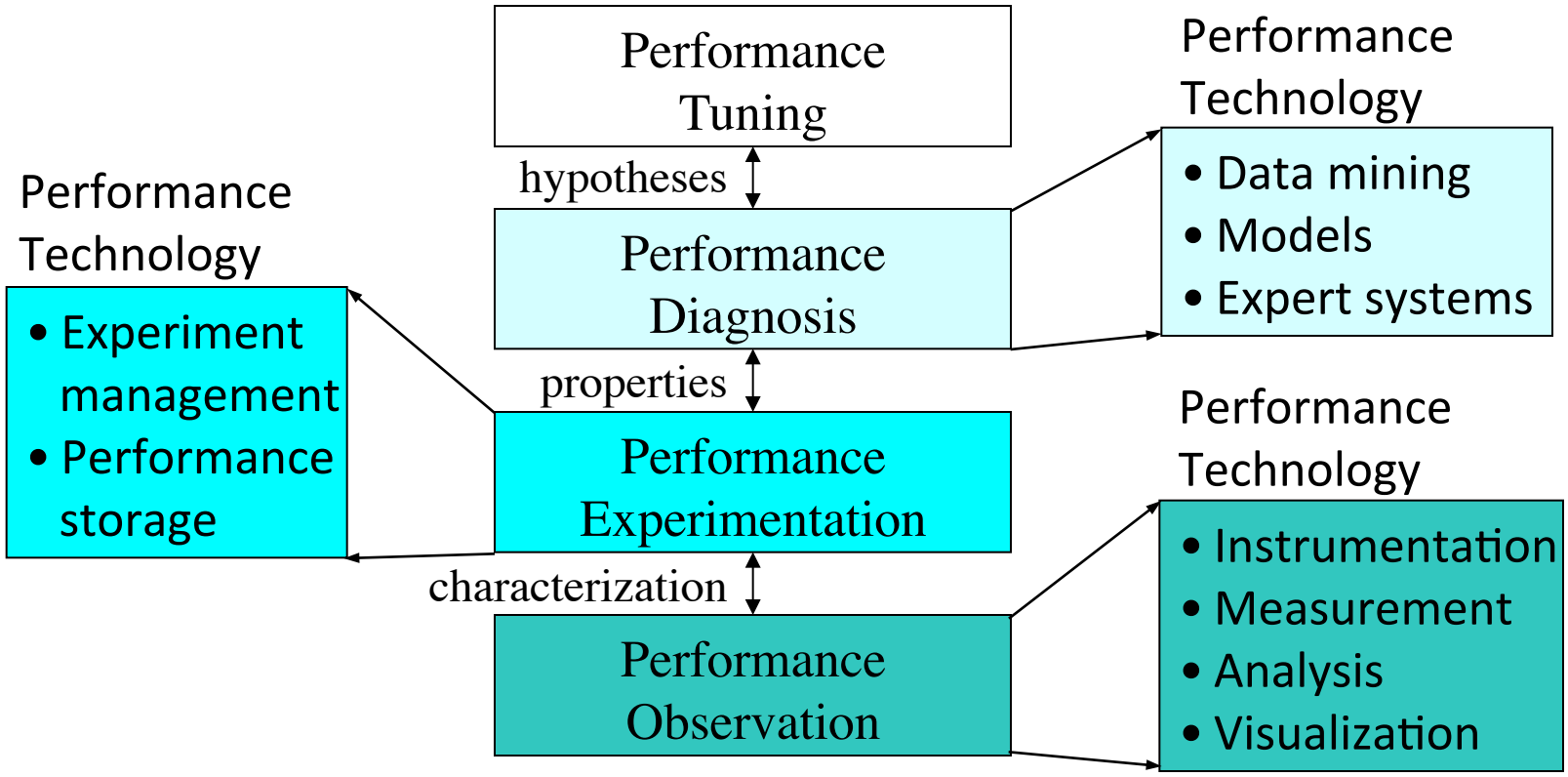
- Data distributed across processes
  - Not logically shared

Parallel Performance Engineering

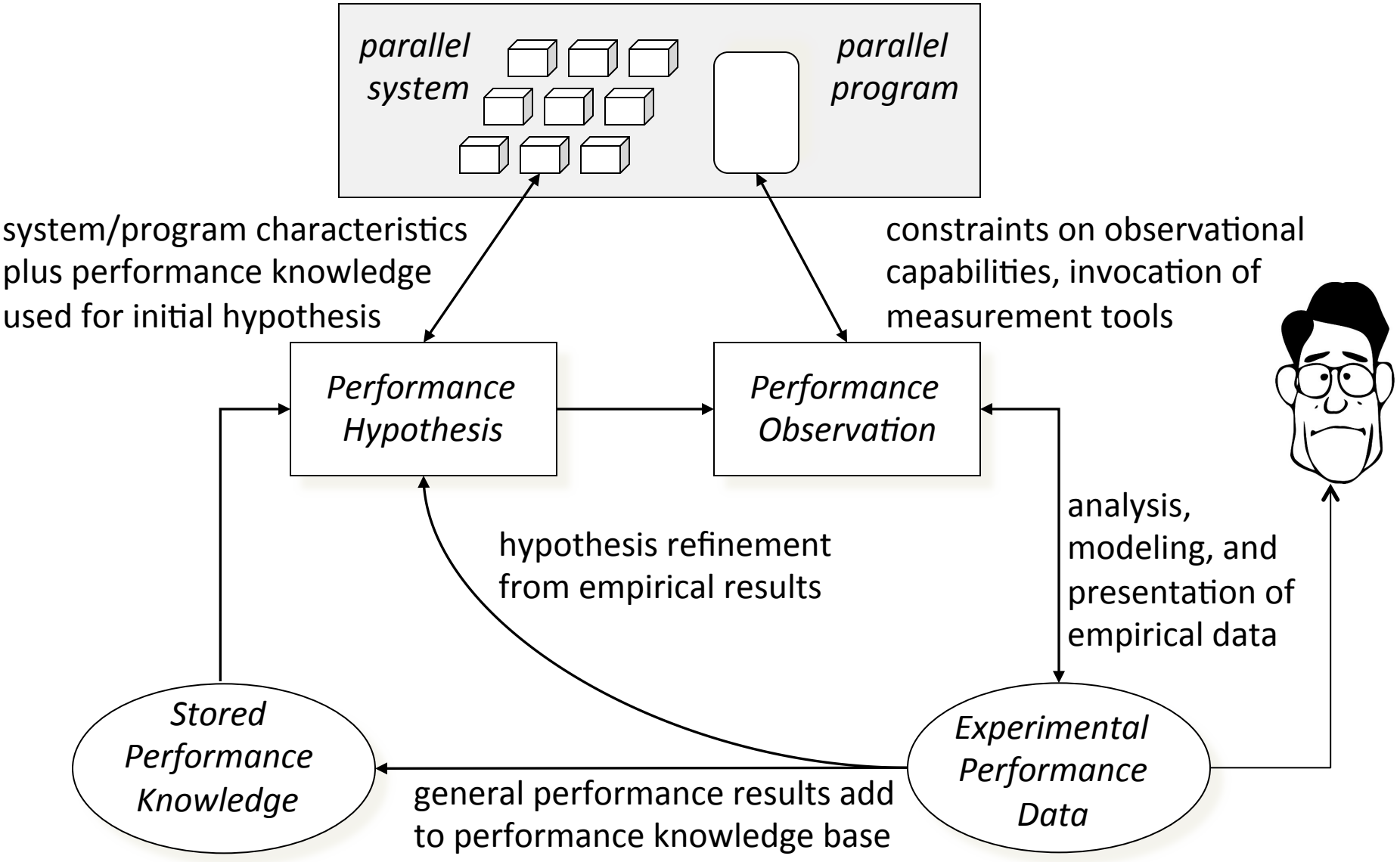
- Scalable, optimized applications deliver HPC promise
- Optimization through performance engineering
  - Understand performance complexity and inefficiencies
  - Tune application to run optimally on high-end machines
- How to make the process effective and productive?
- What performance technology (tools) should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
- Parallel performance tools have been driven by parallel system advances and increased complexity of parallel computation and execution

Parallel Performance Engineering Process

- Traditionally an empirically-based approach
  - observation ⇔ experimentation ⇔ diagnosis ⇔ tuning
- Performance technology developed for each level



Parallel Performance Diagnosis



Performance Expectations

- Context for understanding the performance behavior of the application and system
- Traditional performance expectations are implicit
  - Users must compare to absolute (*peak*) performance
  - Peak measures provide an absolute upper bound
    - rarely is peak performance reached (5-10% is typical)
- How do we know if a parallel program is performing well or performing poorly?
- Performance tools should incorporate better performance expectation to be effective
  - These can come from more knowledge about parallel algorithms, programming environments, HW, systems

Performance Problem Solving Goals

- Answer questions at multiple levels of interest
  - High-level performance data spanning dimensions
    - machine, applications, code revisions, data sets
    - examine broad performance trends
  - Data from low-level measurements
    - use to predict application performance
- Discover general correlations
  - Performance and features of external environment
  - Identify primary performance factors
- Benchmarking analysis for application prediction
- Workload analysis for machine assessment