High Performance Programming on a Single Processor: Memory Hierarchies

Katherine Yelick
yelick@cs.berkeley.edu
http://www.cs.berkeley.edu/~yelick/cs267

Outline

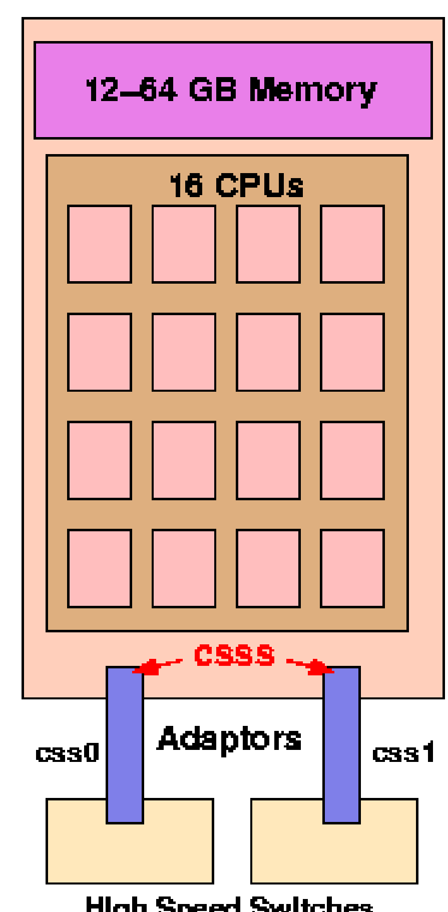
- Goal of parallel computing
  - Solve a problem on a parallel machine that is impractical on a serial one
  - How long does/will the problem take on P processors?
- Quick look at parallel machines
- Understanding parallel performance
  - Speedup: the effectiveness of parallelism
  - Limits to parallel performance
- Understanding serial performance
  - Parallelism in modern processors
  - Memory hierarchies

Microprocessor revolution

- Moore's law in microprocessor performance made desktop computing in 2000 what supercomputing was in 1990
- Massive parallelism has changed the high end
  - From a small number of very fast (vector) processors to
  - A large number (hundreds or thousands) of desktop processors
- Use the fastest "commodity" workstations as building blocks
  - Sold in enough quantity to make them inexpensive
  - Start with the best performance available (at a reasonable price)
- Today, most parallel machines are clusters of SMPs:
  - An SMP is a tightly couple shared memory multiprocessor
  - A cluster is a group of this connected by a high speed network

A Parallel Computer Today: NERSC-3 Vital Statistics

- 5 Teraflop/s Peak Performance – 3.05 Teraflop/s with Linpack
  - 208 nodes, 16 CPUs per node at 1.5 Gflop/s per CPU
- 4.5 TB of main memory
  - 140 nodes with 16 GB each, 64 nodes with 32 GBs, and 4 nodes with 64 GBs.
- 40 TB total disk space
  - 20 TB formatted shared, global, parallel, file space; 15 TB local disk for system usage
- Unique 512 way Double/Single switch configuration



Performance Levels (for example on NERSC-3)

- Peak advertised performance (PAP): 5 Tflop/s
- LINPACK: 3.05 Tflop/s
- Gordon Bell Prize, application performance : 2.46 Tflop/s
  - Material Science application at SC01
- Average sustained applications performance: ~0.4 Tflop/s
  - Less than 10% peak!

Millennium and CITRIS

- Millennium Central Cluster
  - 99 Dell 2300/6350/6450 Xeon Dual/Quad:
  - 332 processors total
  - Total: 211GB memory, 3TB disk
  - Myrinet 2000 + 1000Mb fiber ethernet
- CITRIS Cluster 1: 3/2002 deployment
  - 4 Dell Precision 730 Itanium Duals: 8 processors
  - Total: 20 GB memory, 128GB disk
  - Myrinet 2000 + 1000Mb copper ethernet
- CITRIS Cluster 2: 2002-2004 deployment
  - ~128 Dell McKinley class Duals: 256 processors
  - Total: ~512GB memory, ~8TB disk
  - Myrinet 2000 (subcluster) + 1000Mb copper ethernet
  - ~32 nodes available now

Outline

- Goal of parallel computing
  - Solve a problem on a parallel machine that is impractical on a serial one
  - How long does/will the problem take on P processors?
- Quick look at parallel machines
- Understanding parallel performance
  - Speedup: the effectiveness of parallelism
  - Limits to parallel performance
- Understanding serial performance
  - Parallelism in modern processors
  - Memory hierarchies

Speedup

- The *speedup* of a parallel application is

$$Speedup(p) = Time(1)/Time(p)$$

- Where
  - Time(1) = execution time for a single processor and
  - Time(p) = execution time using p parallel processors
- If Speedup(p) = p we have *perfect speedup* (also called *linear scaling*)

- As defined, speedup compares an application with itself on one and on p processors, but it is more useful to compare
  - The execution time of the best serial application on 1 processor

versus

  - The execution time of best parallel algorithm on p processors

Efficiency

- The *parallel efficiency* of an application is defined as

  Efficiency(p) = Speedup(p)/p

  - Efficiency(p) <= 1
  - For perfect speedup Efficiency (p) = 1
- We will rarely have perfect speedup.
  - Lack of perfect parallelism in the application or algorithm
  - Imperfect load balancing (some processors have more work)
  - Cost of communication
  - Cost of contention for resources, e.g., memory bus, I/O
  - Synchronization time
- Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

Superlinear Speedup

Question: can we find "*superlinear*" speedup, that is

Speedup(p) > p ?

- Choosing a bad "baseline" for T(1)
  - Old serial code has not been updated with optimizations
  - Avoid this, and always specify what your baseline is
- Shrinking the problem size per processor
  - May allow it to fit in small fast memory (cache)
- Application is not deterministic
  - Amount of work varies depending on execution order
  - Search algorithms have this characteristic

Amdahl's Law

- Suppose only part of an application runs in parallel
- Amdahl's law
  - Let s be the fraction of work done serially,
  - So (1-s) is fraction done in parallel
  - What is the maximum speedup for P processors?

Speedup(p) = T(1)/T(p)

T(p) = (1-s)*T(1)/p +s*T(1)

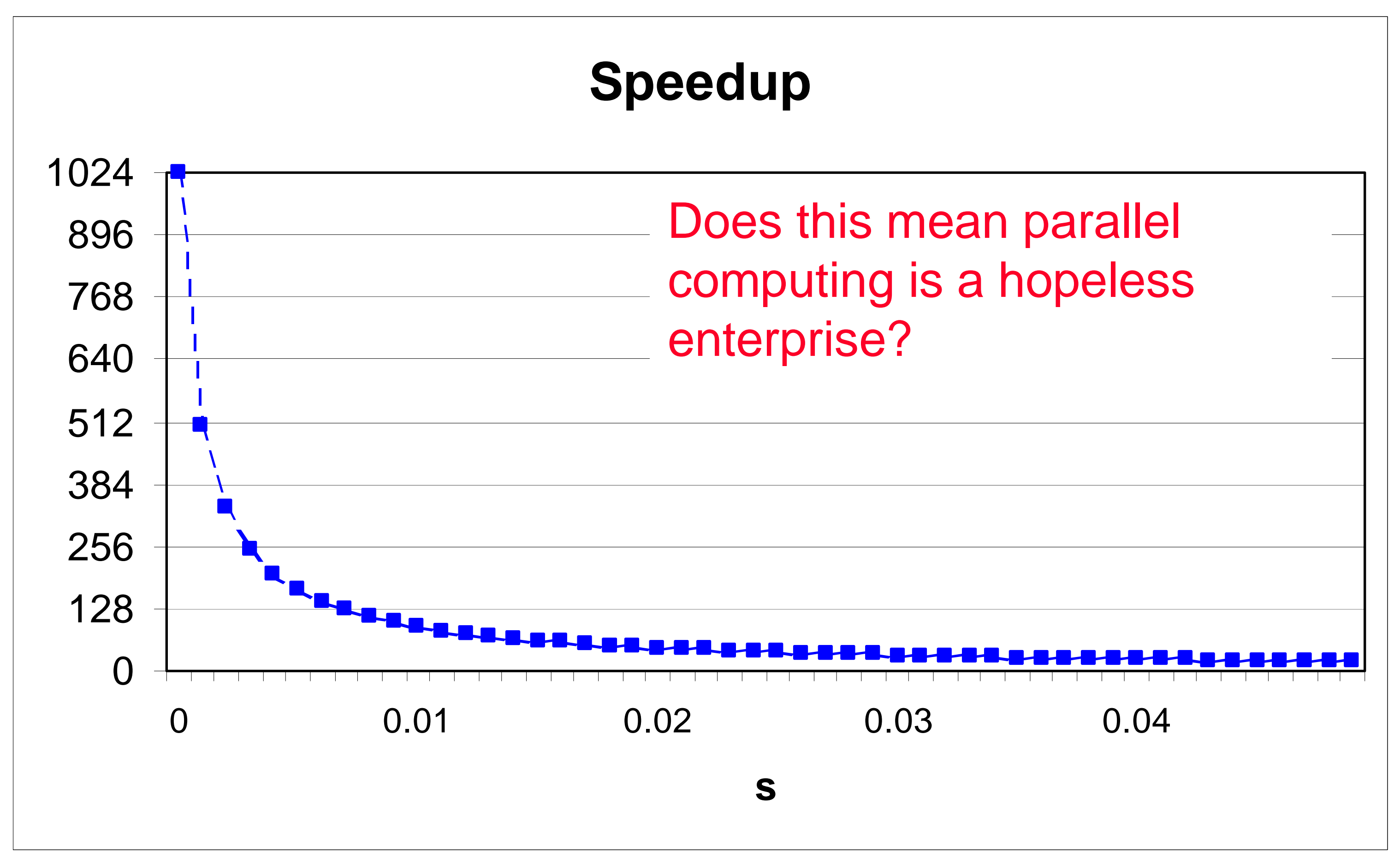
= T(1)*((1-s) + p*s)/p

Speedup(p) = p/(1 + (p-1)*s)

assumes perfect speedup for parallel part

Even if the parallel part speeds up perfectly, we may be limited by the sequential portion of code.

Amdahl's Law (for 1024 processors)



See: Gustafson, Montry, Benner, "Development of Parallel Methods for a 1024 Processor Hypercube", SIAM J. Sci. Stat. Comp. 9, No. 4, 1988, pp.609.

Scaled Speedup

- Speedup improves as the problem size grows
  - Among other things, the Amdahl effect is smaller
- Consider
  - scaling the problem size with the number of processors (add problem size parameter, n)
  - for problem in which running time scales linearly with the problem size: T(1,n) = T(1)*n
  - let n=p (problem size on p processors increases by p)

$$\text{ScaledSpeedup}(p,n) = T(1,n)/T(p,n)$$

$$T(p,n) = (1-s)*n*T(1,1)/p + s*T(1,1)$$

$$= (1-s)*T(1,1) + s*T(1,1) = T(1,1)$$

$$\text{ScaledSpeedup}(p,n) = n = p$$

assumes serial work does not grow with n

Scaled Efficiency

- Previous definition of *parallel efficiency* was

$$\text{Efficiency}(p) = \text{Speedup}(p)/p$$

- We often want to scale problem size with the number of processors, but scaled speedup can be tricky
  - Previous definition depended on a linear work in problem size
- May use alternate definition of efficiency that depends on a notion of throughput or rate, R(p):
  - Floating point operations per second
  - Transactions per second
  - Strings matches per second
- Then

$$\text{Efficiency}(p) = R(p)/(R(1)*p)$$

- May use a different problem size for R(1) and R(p)

Three Definitions of Efficiency: Summary

- People use the word "efficiency" in many ways
- Performance relative to advertised machine peak

  Flop/s in application / Max Flops/s on the machine

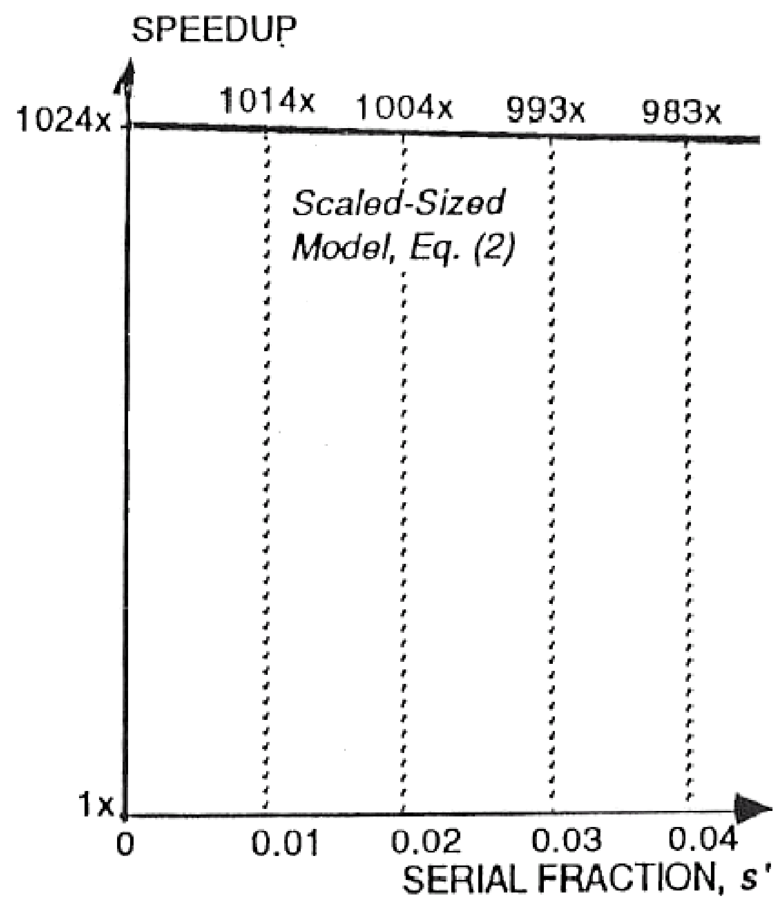
  - Integer, string, logical or other operations could be used, but they should be a machine-level instruction
- Efficiency of a fixed problem size

  Efficiency(p) = Speedup(p)/p
- Efficiency of a scaled problem size

  Efficiency(p) = R(p)/(R(1)*p)

- All of these may be useful in some context
- Always make it clear what you are measuring

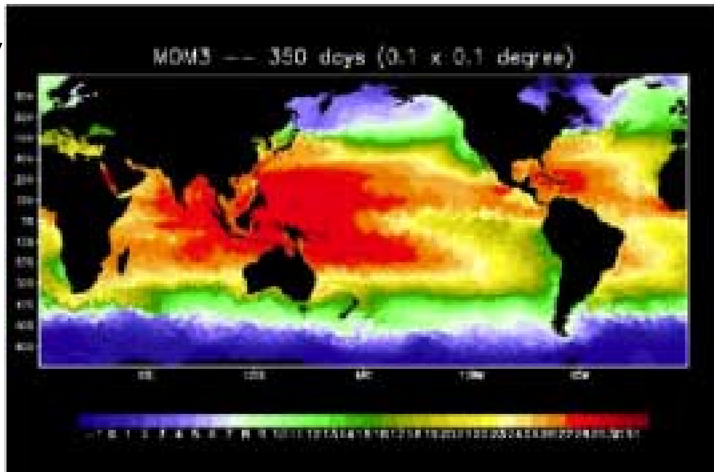
Scaled Speedup



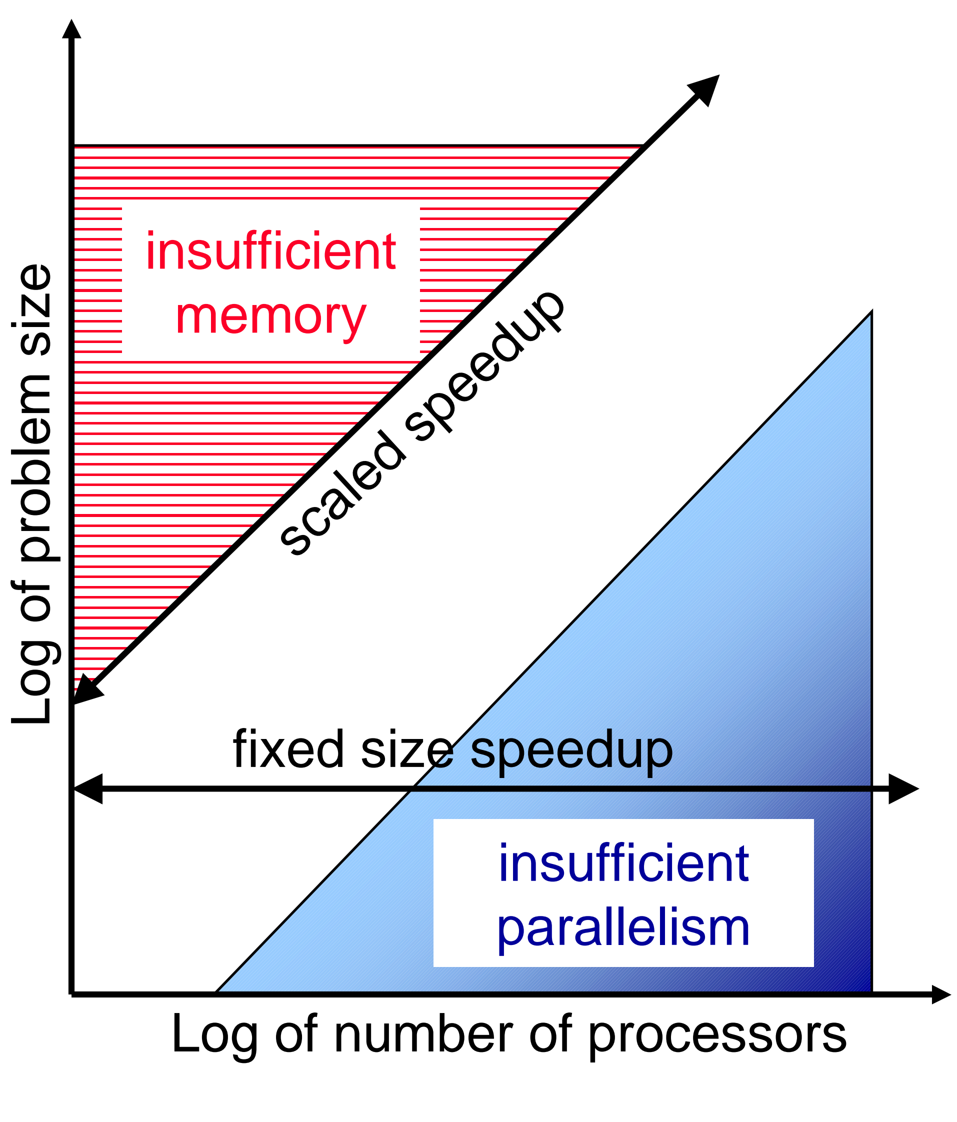
See: Gustafson, Montry, Benner, “Development of Parallel Methods for a 1024 Processor Hypercube”, SIAM J. Sci. Stat. Comp. 9, No. 4, 1988, pp.609.

Limits of Scaling – an example of a current debate

- Test run on global climate model reported on the Earth Simulator sustained performance of about 28 TFLOPS on 640 nodes. The model was an atmospheric global climate model (T1279L96) developed originally by CCSR/NEIS and tuned by ESS.
- This corresponds to scaling down to a 10 km^2 grid
- Many physical modeling assumptions from a 200 km^2 grid don't hold any longer
- The climate modeling community is debating the significance of these results



Performance Limits



Outline

- Goal of parallel computing
  - Solve a problem on a parallel machine that is impractical on a serial one
  - How long does/will the problem take on P processors?
- Quick look at parallel machines
- Understanding parallel performance
  - Speedup: the effectiveness of parallelism
  - Limits to parallel performance
- Understanding serial performance
  - Parallelism in modern processors
  - Memory hierarchies

Principles of Parallel Computing

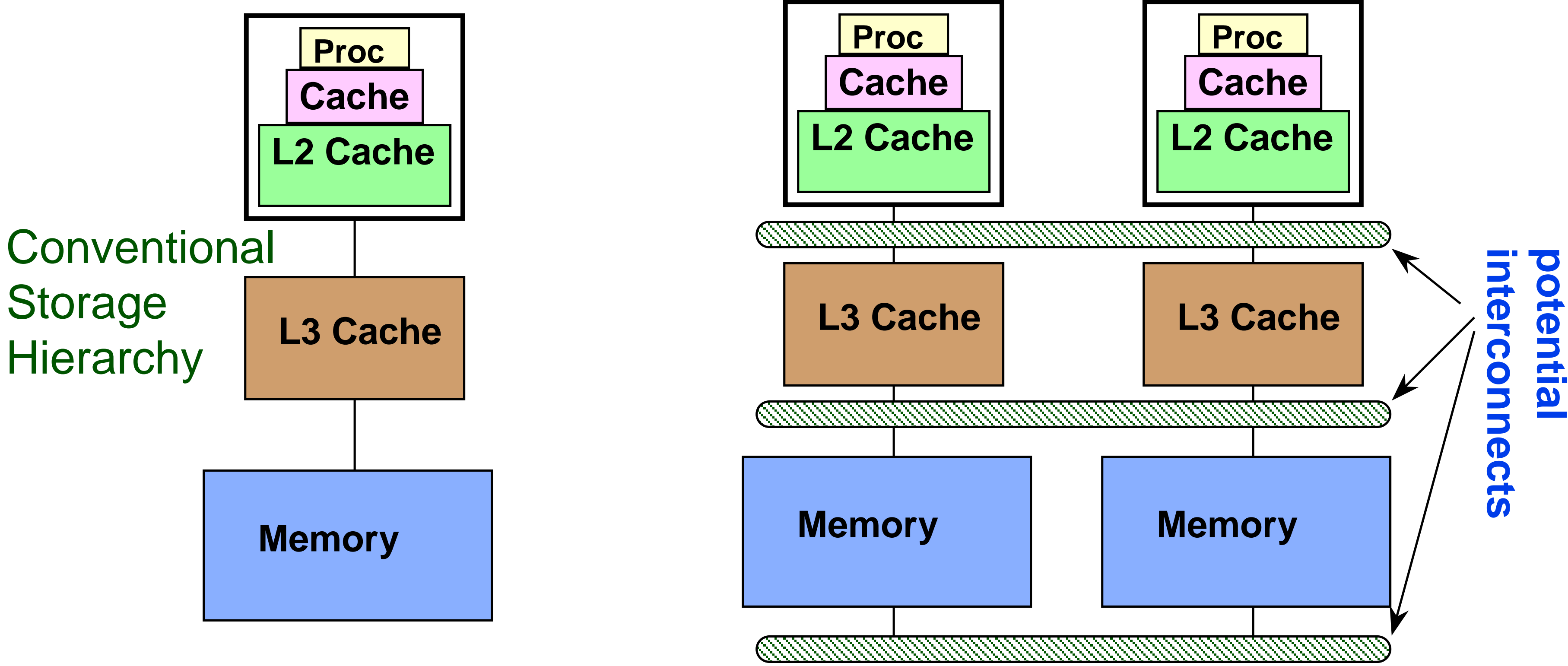
- Speedup, efficiency, and Amdahl's Law
- Finding and exploiting parallelism
- Finding and exploiting data locality
- Load balancing
- Coordination and synchronization
- Performance modeling

All of these things make parallel programming more difficult than sequential programming.

Overhead of Parallelism

- Given enough parallel work, this is the most significant barrier to getting desired speedup.
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (= millions of arithmetic ops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work.

Locality and Parallelism

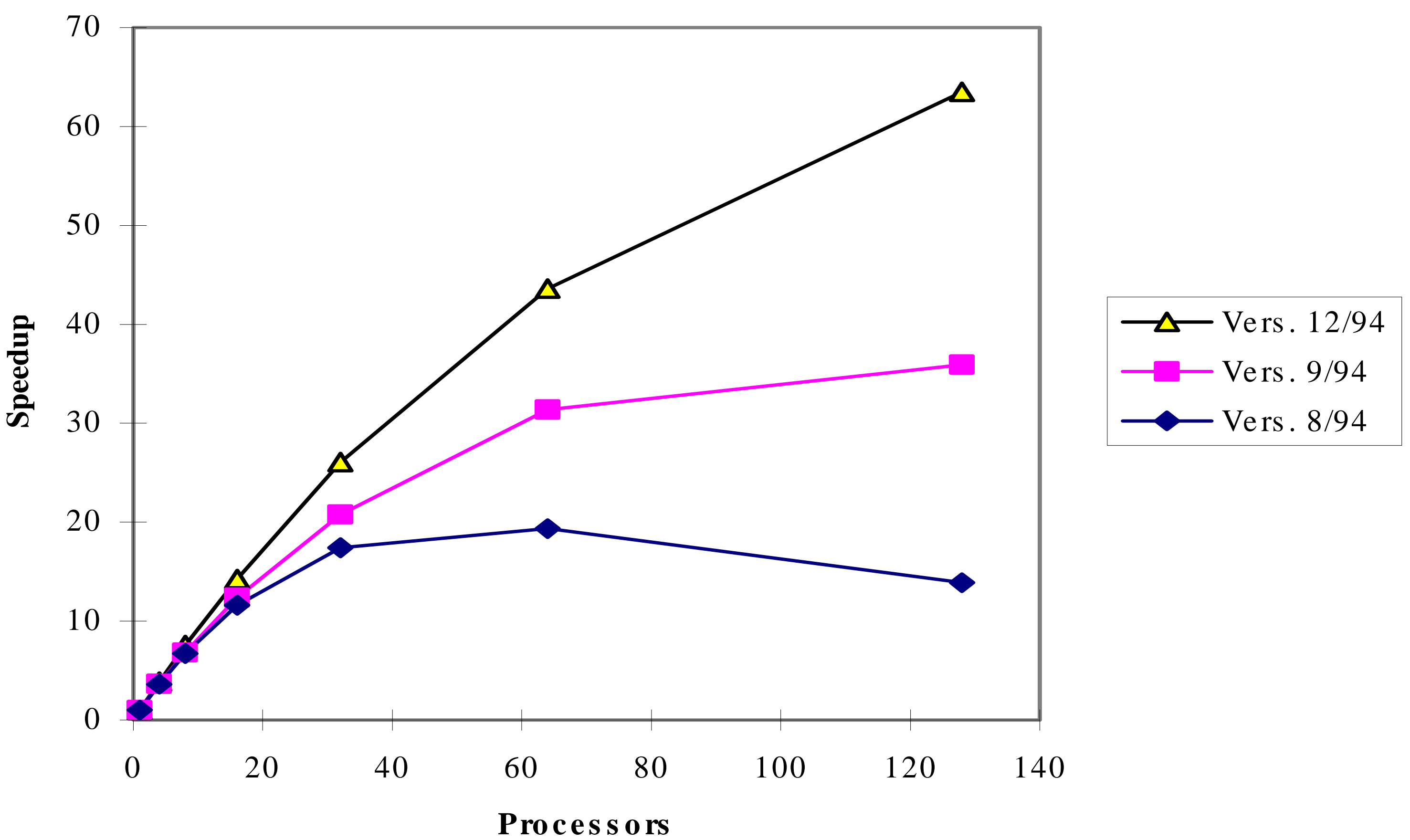


- Large memories are slower; fast memories are small.
- Storage hierarchies are designed to fast on average.
- Parallel processors, collectively, have large, fast memories -- the slow accesses to “remote” data we call “communication”.
- Algorithm should do most work on local data.

Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase).
  - unequal size tasks.
- Examples of the latter
  - adapting to “interesting parts of a domain”.
  - tree-structured computations.
  - fundamentally unstructured problems.
- Algorithm needs to balance load
  - but techniques the balance load often reduce locality

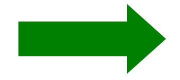
Performance Programming is Challenging



- Speedup(P) = Time(1) / Time(P)
- Applications have "learning curves"

Outline

- Goal of parallel computing
  - Solve a problem on a parallel machine that is impractical on a serial one
  - How long does/will the problem take on P processors?
- Quick look at parallel machines
- Understanding parallel performance
  - Speedup: the effectiveness of parallelism
  - Limits to parallel performance
- Understanding serial performance



  - Parallelism in modern processors
  - Memory hierarchies

Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
  - Exist in the program stack, static region, or heap
- Operations include
  - Read and write (given an address/pointer)
  - Arithmetic and other logical operations
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into "obvious" lower level instructions
  - Hardware executes instructions in order specified by compiler
- Cost
  - Each operations has roughly the same cost (read, write, add, multiply, etc.)

Uniprocessors in the Real World

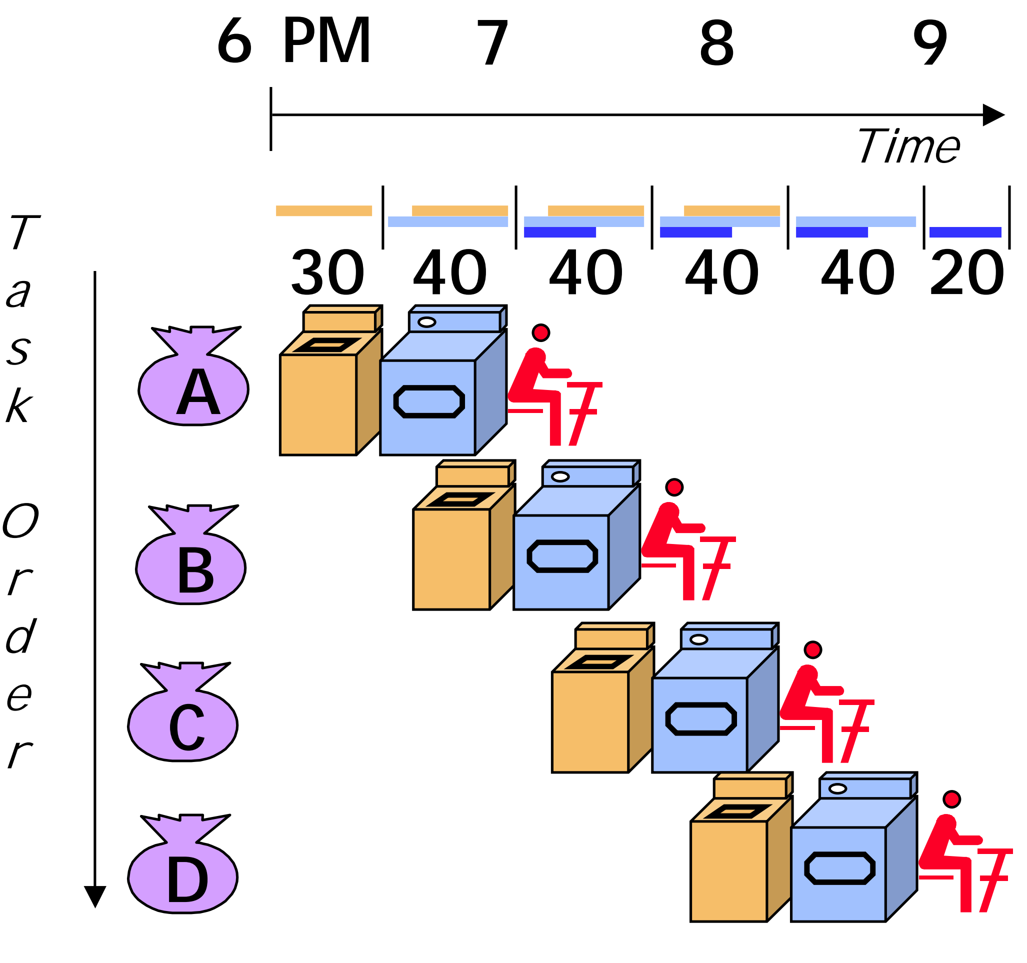
- Real processors have
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs
  - parallelism
    - multiple “functional units” that can run in parallel
    - different orders, instruction mixes have different costs
  - pipelining
    - a form of parallelism, like an assembly line in a factory
- Why is this your problem?

  In theory, compilers understand all of this and can optimize your program; in practice they don’t.

What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry

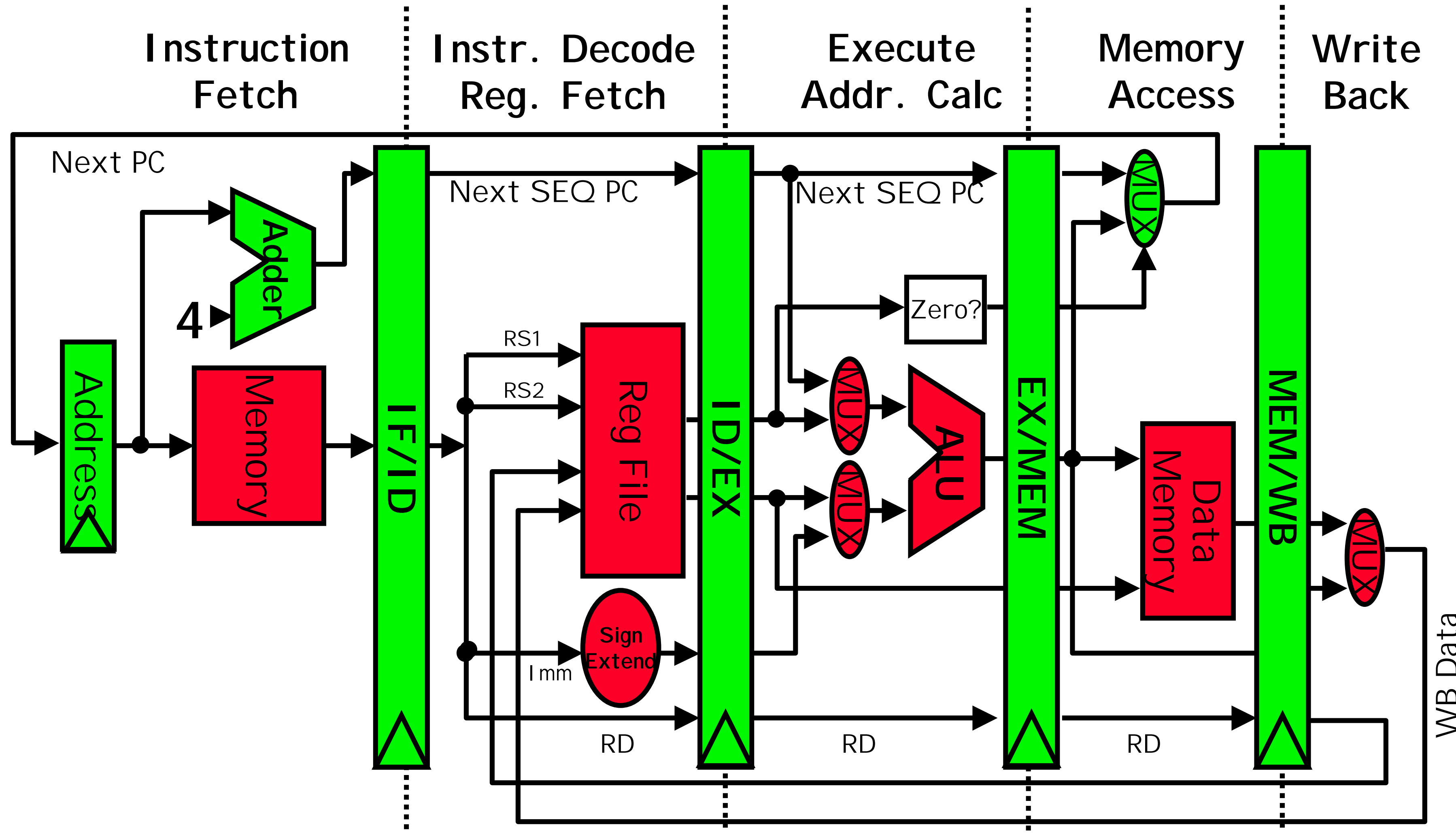
wash (30 min) + dry (40 min) + fold (20 min)



- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.3 hours
- Pipelining helps throughput, but not latency
- Pipeline rate limited by slowest pipeline stage
- Potential speedup = Number pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup

Example: 5 Steps of MIPS Datapath

Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy



- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle

Limits to Instruction Level Parallelism (ILP)

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
- The hardware and compiler will try to reduce these:
  - Reordering instructions, multiple issue, dynamic branch prediction, speculative execution…
- You can also enable parallelism by careful coding

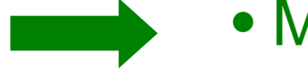
Dependences (Data Hazards) Limit Parallelism

- A dependence or data hazard is one of the following:
  - true of flow dependence:
    - a writes a location that b later reads
    - (read-after write or RAW hazard)
  - anti-dependence
    - a reads a location that b later writes
    - (write-after-read or WAR hazard)
  - output dependence
    - a writes a location that b later writes
    - (write-after-write or WAW hazard)

true	anti	output
a = = a	= a a =	a = a =

Outline

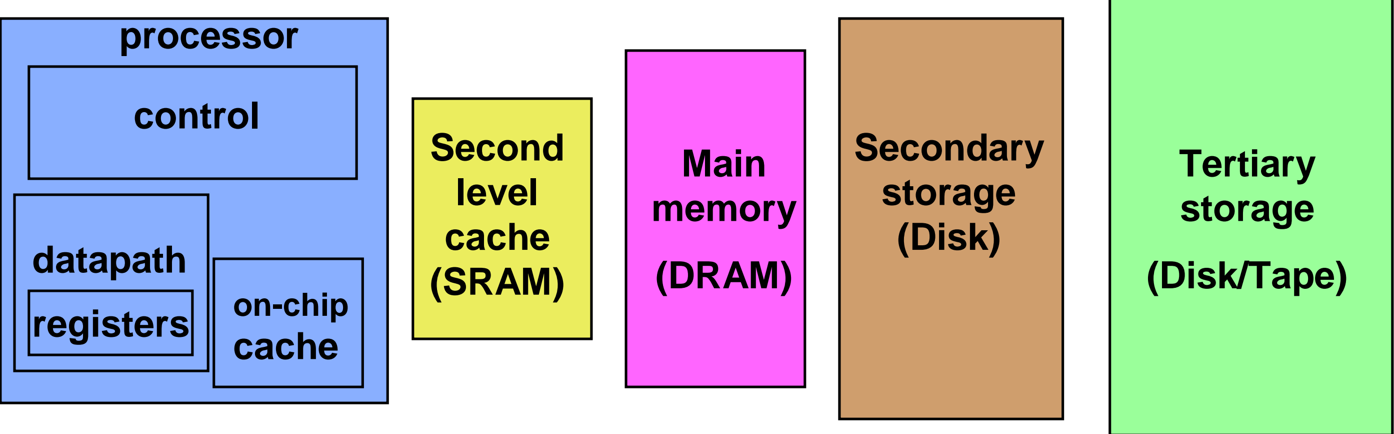
- Goal of parallel computing
    - Solve a problem on a parallel machine that is impractical on a serial one
    - How long does/will the problem take on P processors?
- Quick look at parallel machines
- Understanding parallel performance
    - Speedup: the effectiveness of parallelism
    - Limits to parallel performance
- Understanding serial performance
    - Parallelism in modern processors



    - Memory hierarchies

Memory Hierarchy

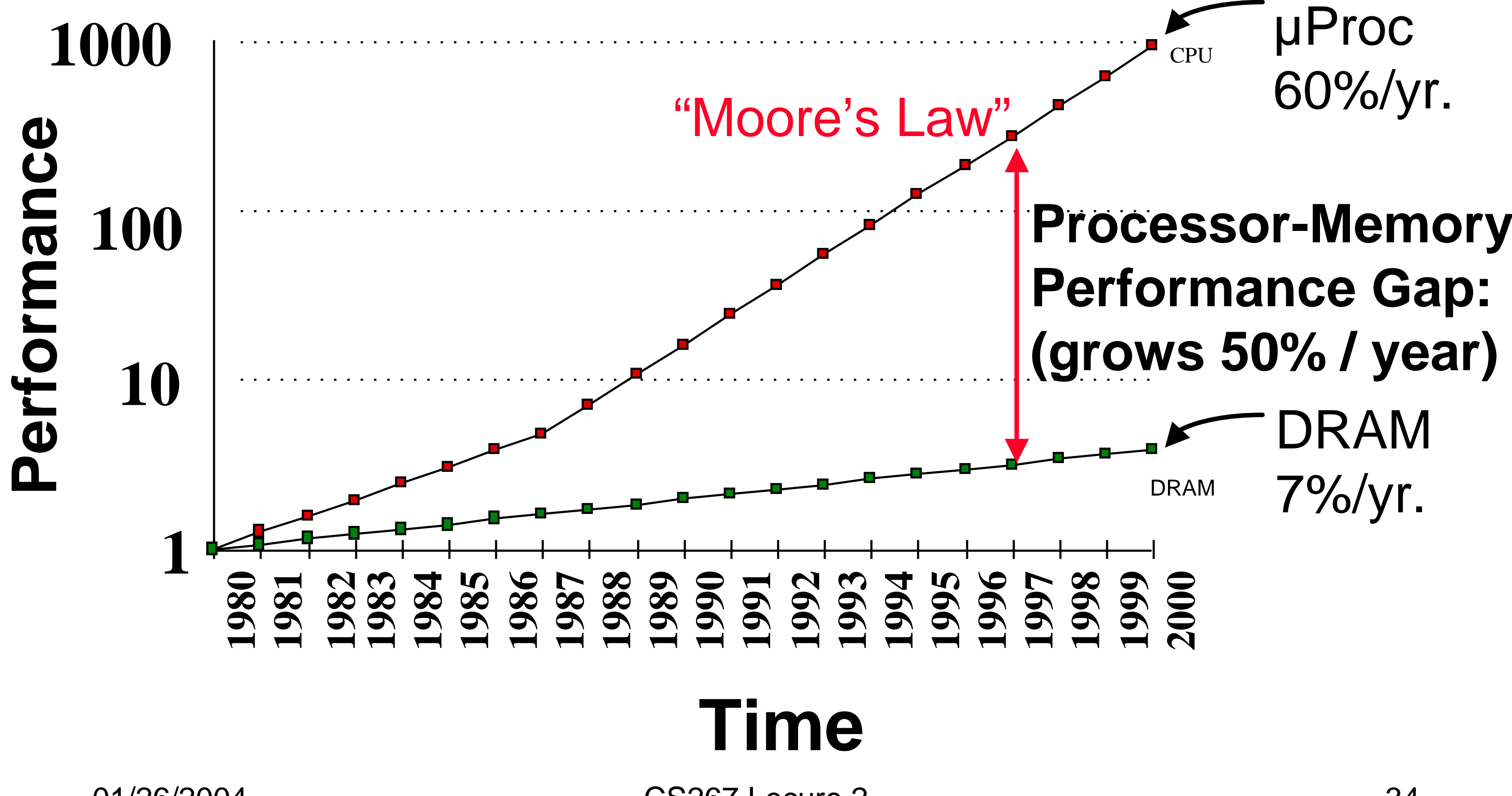
- Most programs have a high degree of locality in their accesses
  - spatial locality: accessing things nearby previous accesses
  - temporal locality: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality



Speed	1ns	10ns	100ns	10ms	10sec
Size	B	KB	MB	GB	TB

Processor-DRAM Gap (latency)

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory



Approaches to Handling Memory Latency

- Approach to address the memory latency problem
  - Build faster memories: bandwidth has improved much more than latency
  - Eliminate memory operations by saving values in small, fast memory and reuse them (need temporal locality)
  - Take advantage of bandwidth by getting a chunk of memory and save it in small fast memory (need spatial locality)
  - Take advantage of bandwidth by allowing processor to issue multiple reads to the memory system at once (requires concurrency in the instruction stream)

Little's Law

Principle (Little's Law): the relationship of a production system in steady state is:

Inventory = Throughput × Flow Time

For parallel computing, this means:

Required concurrency = Bandwidth x Latency

Example: 1000 processor system, 1 GHz clock (1ns), 100 ns memory latency, 100 words of memory in data paths between CPU and memory at any given time.

- Main memory bandwidth is:
  - ~ 1000 x 100 words x 10^9/s = 10^{14} words/sec.
- To achieve full performance, an application needs:
  - ~ 10^{-7} x 10^{14} = 10^7 way concurrency

Cache Basics

- Cache hit: in-cache memory access—cheap
- Cache miss: non-cached memory access—expensive
- Consider a tiny cache (for illustration only)

Address Pattern	Data (4 Bytes)
X000	101000 through 101001
X010	001010 through 001011
X100	111100 through 111101
X110	110110 through 110111

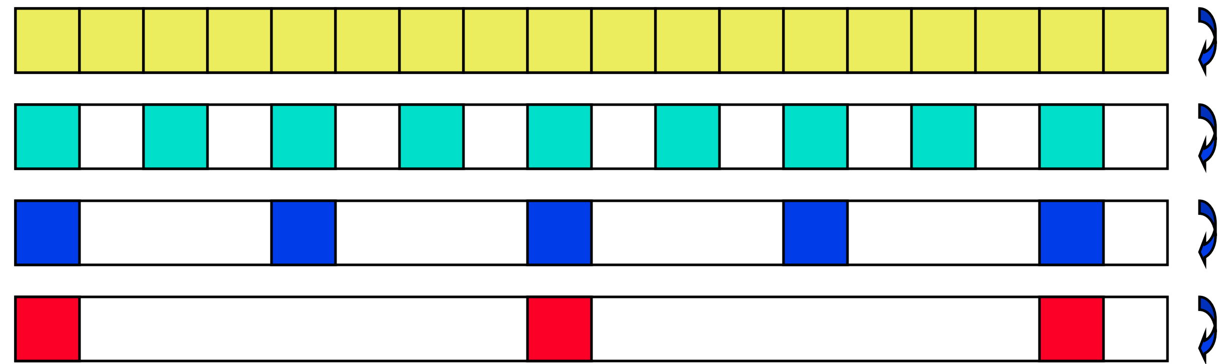
- Cache line length: # of bytes loaded together in one entry
  - 2 in example
- Associativity
  - direct-mapped: only 1 address (line) in a given range in cache
  - *n*-way: 2 or more lines with different addresses exist

Why Have Multiple Levels of Cache?

- On-chip vs. off-chip
  - On-chip caches are faster, but limited in size
- A large cache has expenses
  - Hardware to check addresses in cache can get expensive
  - Associativity, which gives a more general set of data in cache, is also expensive
- Some examples:
  - Cray T3E eliminated one cache to speed up misses
  - IBM uses a level of cache as a "victim cache" which is cheaper
- There are other levels of the memory hierarchy
  - Register, pages (virtual memory), …
  - And it isn't always a hiearchy

Experimental Study of Memory (Membench)

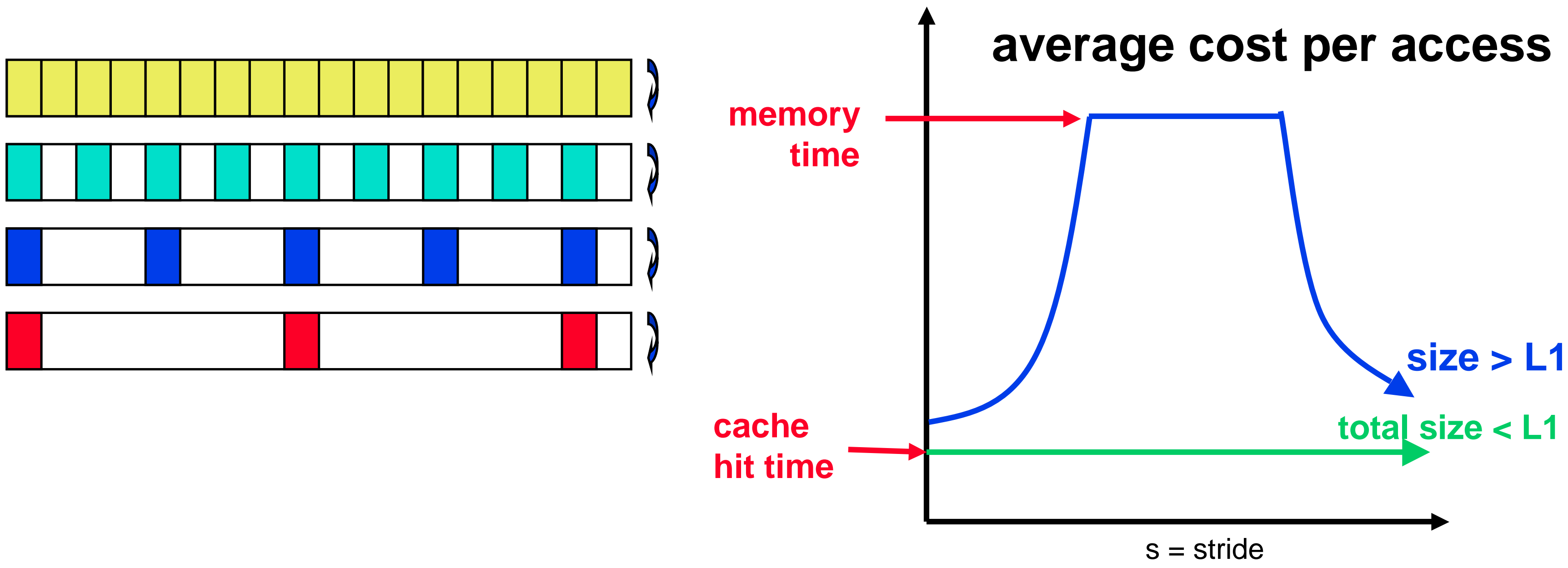
- Microbenchmark for memory system performance



- time the following program for each size(A) and stride s
  (repeat to obtain confidence and mitigate timer resolution)

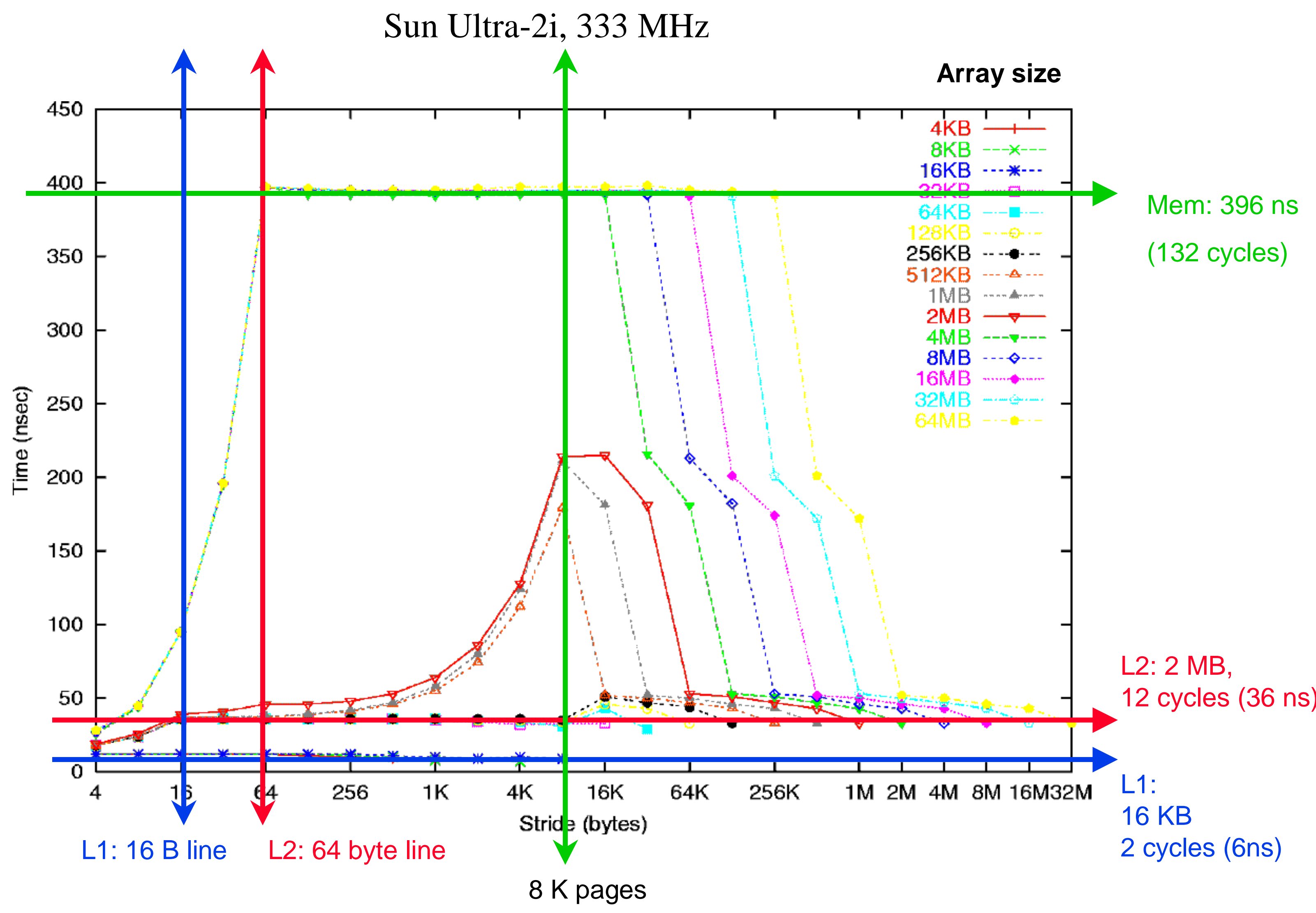
```
for array A of size from 4KB to 8MB by 2x
    for stride s from 8 Bytes (1 word) to size(A)/2 by 2x
        for i from 0 to size by s
            load A[i] from memory (8 Bytes)
```

Membench: What to Expect



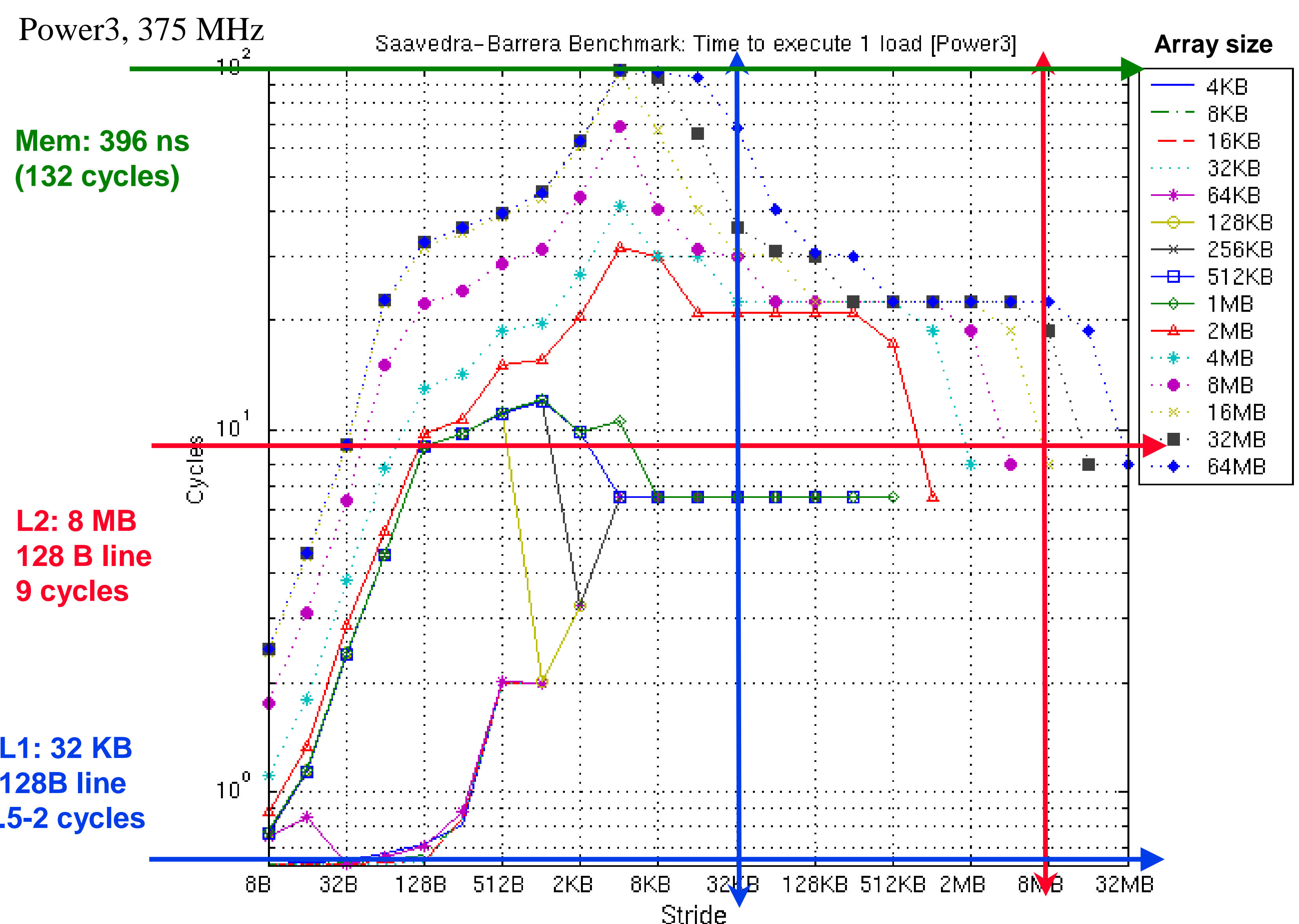
- Consider the average cost per load
  - Plot one line for each array size, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most ¼ miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
  - Picture assumes only one level of cache
  - Values have gotten more difficult to measure on modern procs

Memory Hierarchy on a Sun Ultra-2i



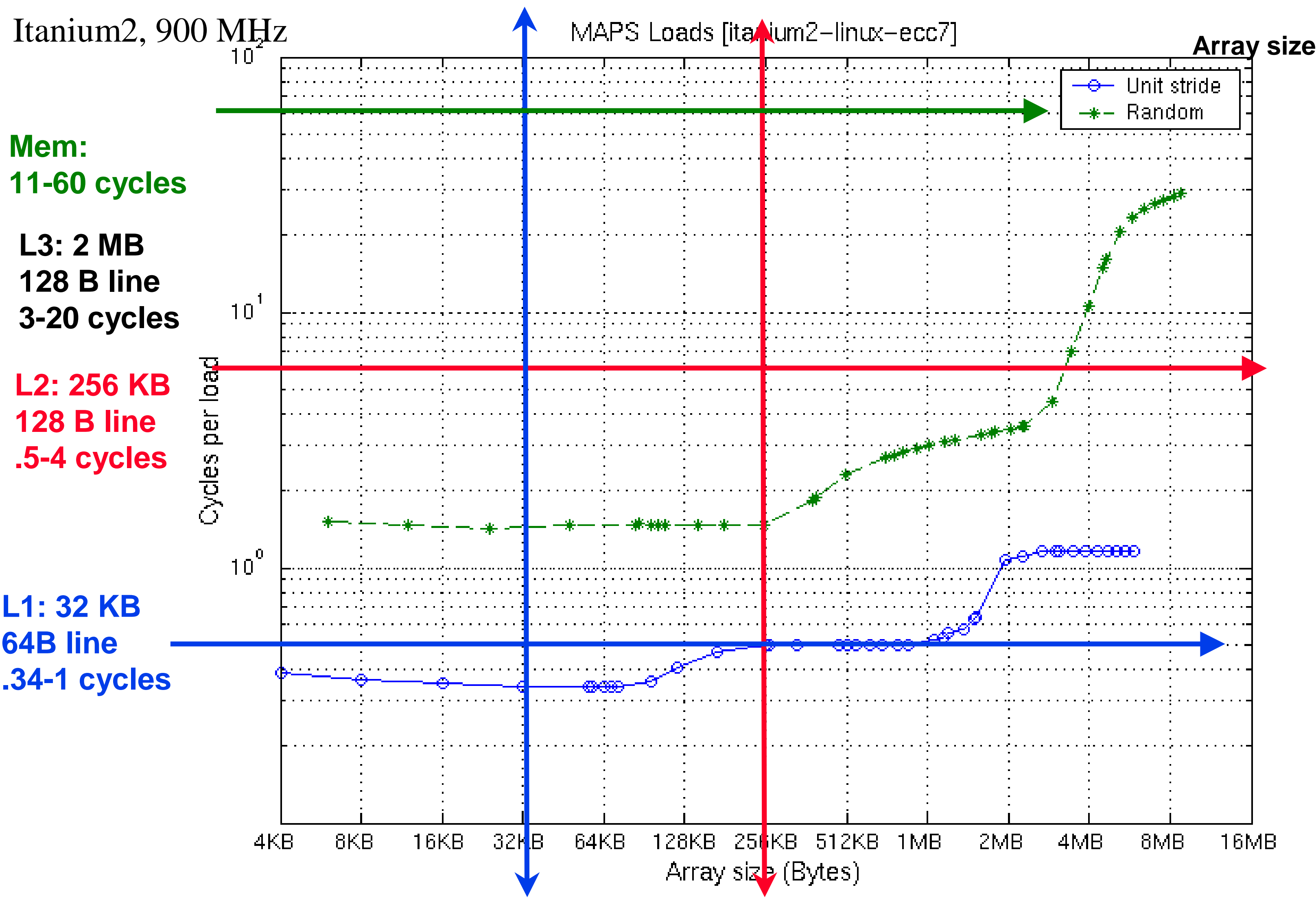
See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details

Memory Hierarchy on a Power3 (Seaborg)



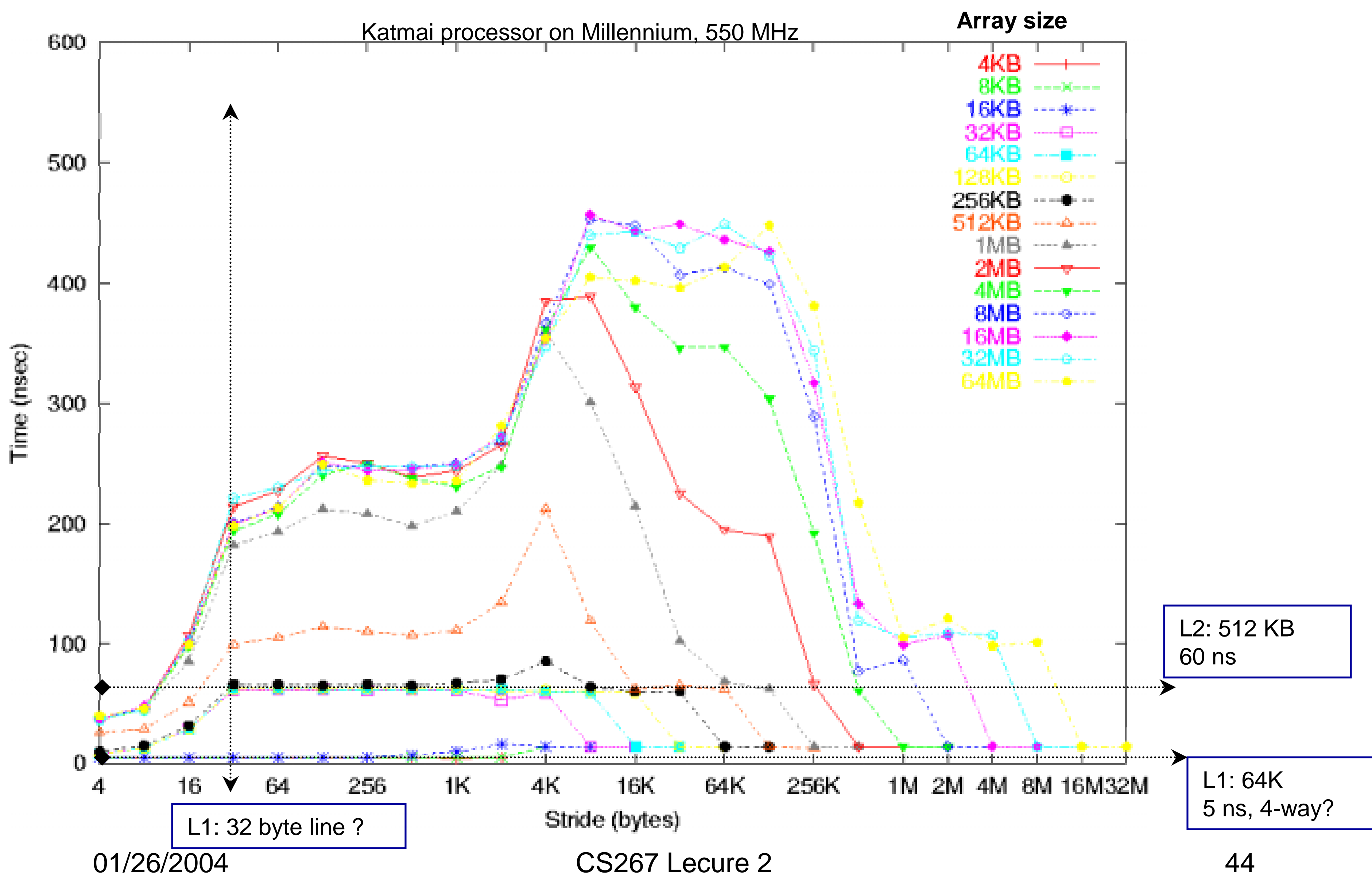
0

Memory Performance on Itanium 2 (CITRIS)



Uses MAPS Benchmark: http://www.sdsc.edu/PMaC/MAPs/maps.html

Memory Hierarchy on a Pentium III



Lessons

- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance significantly
  - To write fast programs, need to consider architecture
  - We would like simple models to help us design efficient algorithms
  - Is this possible?
- We will illustrate with a common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache