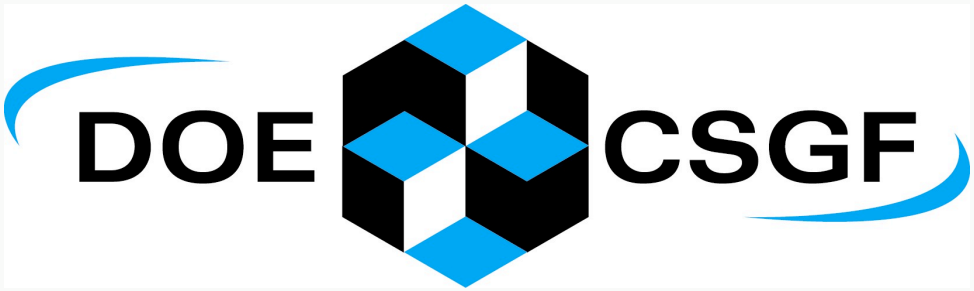



# Computer Architecture

William Gropp


# Why Should You Learn About Computer Architecture

- Lets look at a simple example
- Matrix-matrix multiply
  - Classic example, often used in discussion of compiler optimizations
  - Core of the "HPLinpack" benchmark
  - Simple to express: In Fortran,

```
do i=1, n
   do j=1,n
      c(i,j) = 0
      do k=1,n
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

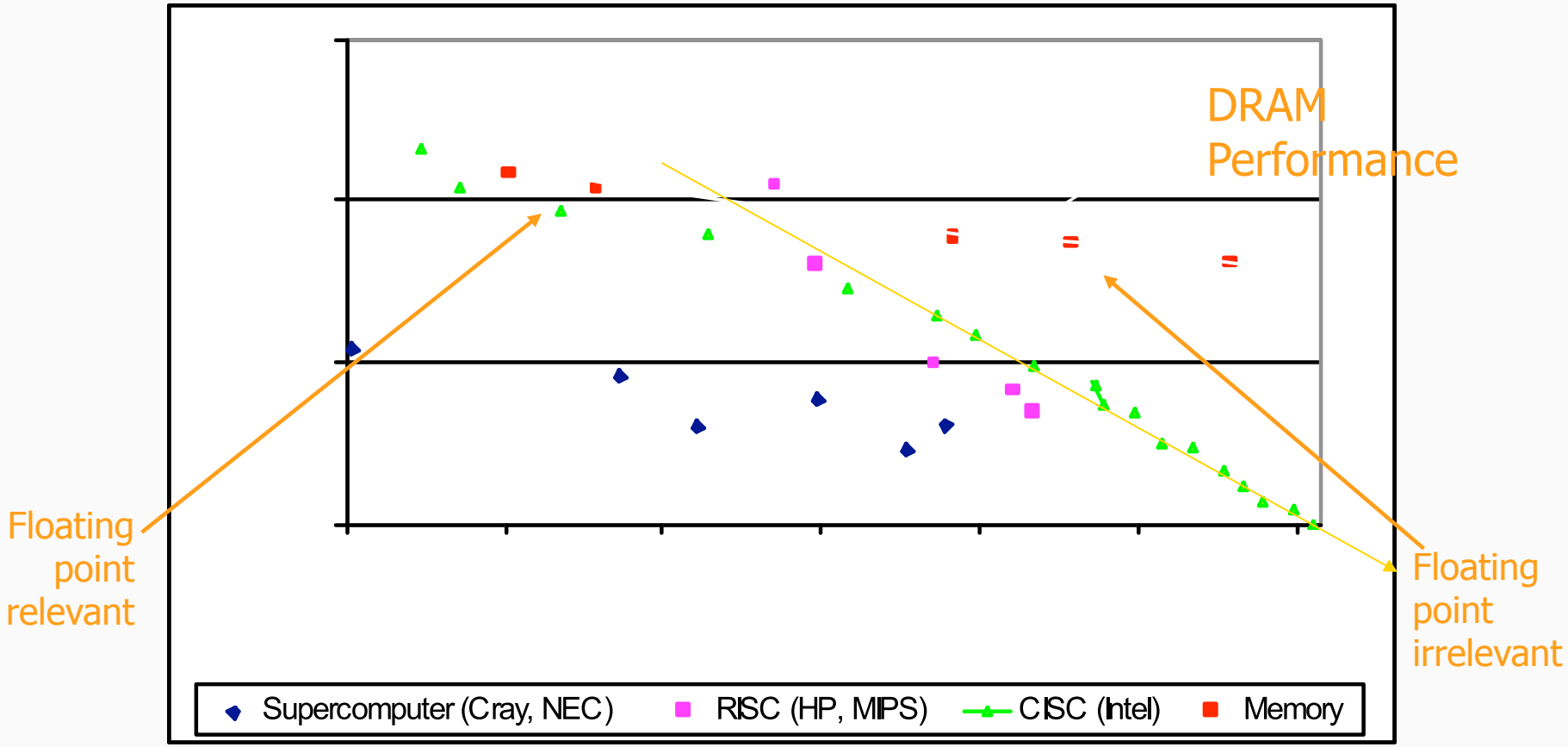

# Performance Estimate

- How fast should this run?
  - Standard complexity analysis in numerical analysis counts floating point operations
  - Our matrix-matrix multiply algorithm has $2n^3$ floating point operations
    - —3 nested loops, each with n iterations
    - —1 multiply, 1 add in each inner iteration
  - For n=100, $2x10^6$ operations, or about 1 msec on a 2GHz processor :)
  - For n=1000, $2x10^9$ operations, or about 1 sec


# The Reality

- N=100
  - 1818 MF (1.1ms)
- N=1000
  - 335 MF (6s)
- What this tells us:
  - Obvious expression of algorithms are not transformed into leading performance.
- Try it yourself! Code on web site

# Why is achieved performance on a single node so poor?

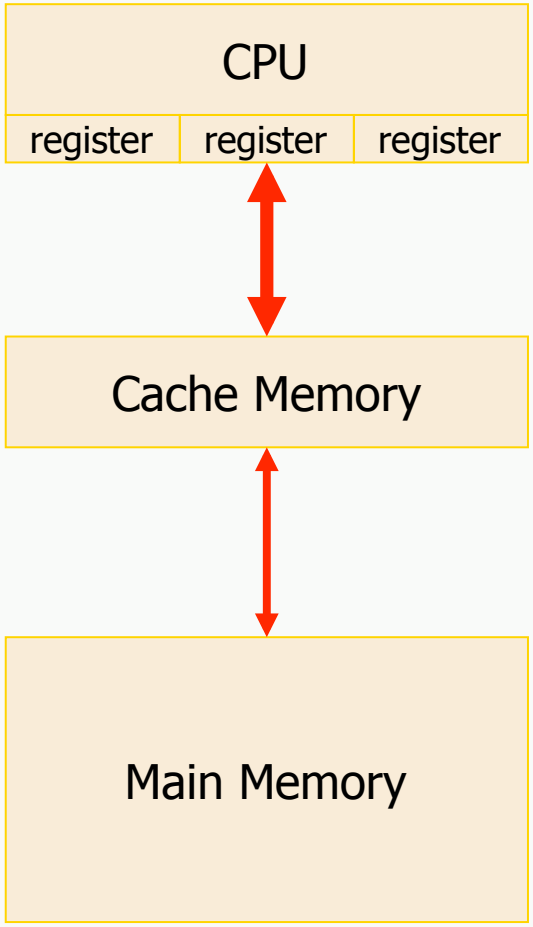



# Trends in Computer Architecture

- Latency to memory will continue to grow relative to (multicore) CPU performance
  - Latency hiding techniques require finding increasing amounts of independent work: Little's law implies
    - —Number of concurrent memory references = Latency * rate
    - —For 1 reference per cycle, this is already 100–1000 concurrent references
- Power dissipation has become a major problem
- As a consequence, clock rates have stopped increasing
  - Most future performance will come from greater parallelism

# Simplified Computer Architecture

- Main memory contains the program data
- Cache memory contains a copy of the main memory data
  - Cache is faster but consumes more space and power
  - Cache items accessed by their address in main memory
- Registers contain working data only
  - Modern CPUs perform most or all operations only on data in register

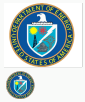



# Simplified Performance Models

- The processor runs in discrete steps, controlled by a *clock*
  - Typical clock rates exceed 2 GHz
- Few operations complete in a single clock period
- One way to describe the operations is with two parameters:
  - Latency: the time it takes to complete a single operation (number of clock ticks or cycles)
  - Bandwidth: the steady-state rate at which operations can be completed
- Example: Memory access
  - A read of a single word from main memory may take 450+ cycles (IBM POWER6)
  - Bandwidth to main memory is GB/s (>4 on POWER6)

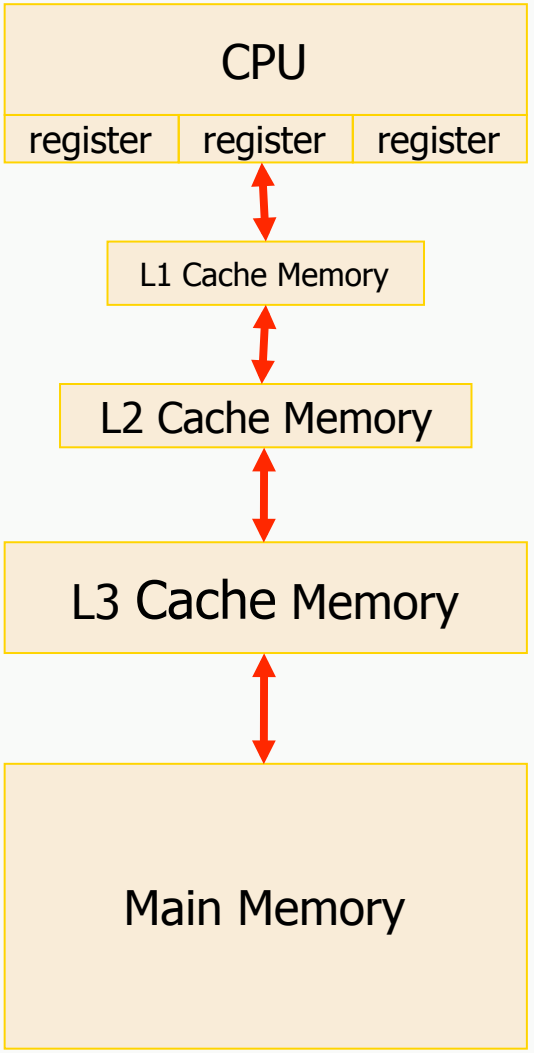

# Some Simple Performance Models

- Clock Rate: Count key operations (such as floating point) and divide by the clock rate
  - Determines the "Peak Performance"
  - Is not a good predictor unless data fits into fastest memory (cache)
- Memory Bandwidth: Count key operations and loads and stores. Use clock rate for operations, memory bus bandwidth for loads and stores
  - Assumes that the memory system design is perfect (the caches always work)
  - Turns out to not be very useful either (we'll see why later)
- Sustained Memory Bandwidth: As Memory Bandwidth, but use measured memory performance
  - Assumes that the cache works to provide the sustained memory system performance
  - Can give us a useful (approximate) upper bound on performance



# Simplified Computer Architecture II

- Main memory contains the program data
- Multiple Cache memories contain a copy of the main memory data
  - Cache is faster but consumes more space and power
  - Cache items accessed by their address in main memory
  - L1 cache is the fastest but has the least capacity
  - L2, L3 provide intermediate performance/size tradeoffs




# Memory Locality

- Typical access times for cache-based systems

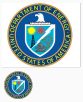
| Register | 1 cycle | Ratio to previous |
|---|---|---|
| L1 Cache | Few cycles | 1-3x |
| L2 Cache | ~ 10 cycles | 3-10x |
| Memory | ~ 250 cycles | 25x |
| Remote Memory | ~2500-5000 cycles | 10-20x |

# Another Example: Reductions

- Consider this code

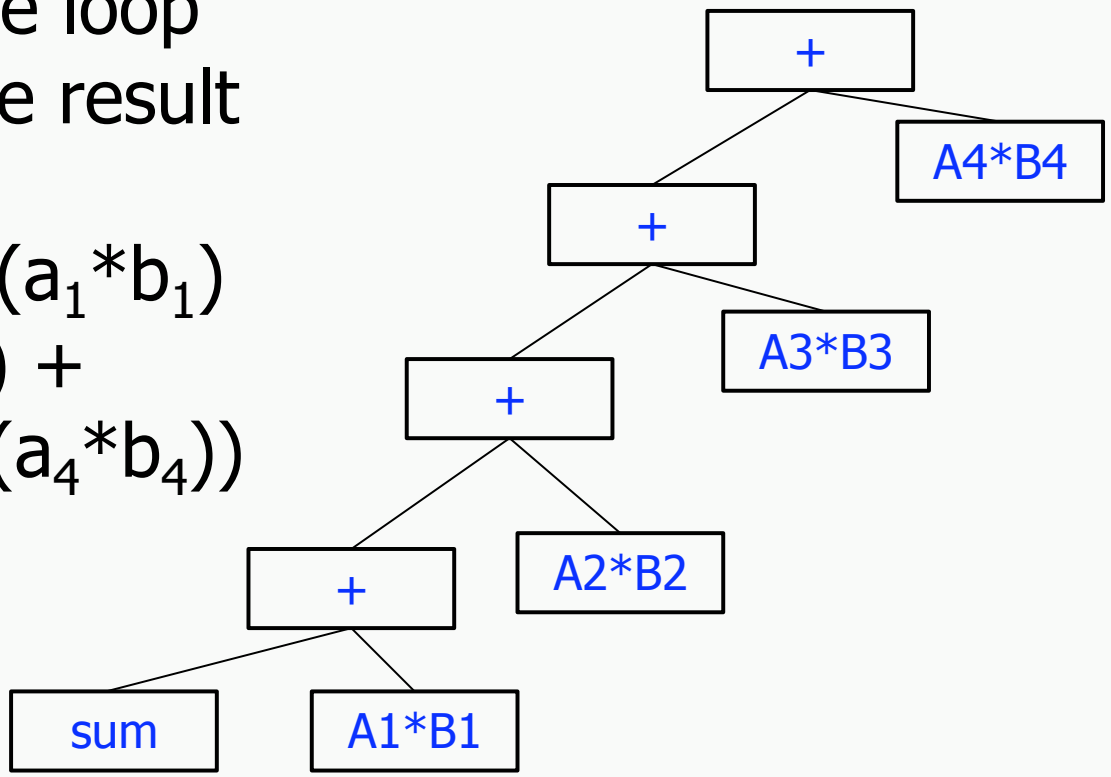
```
do i=1,n
    sum = sum + a(i) * b(i)
enddo
```

- How fast can this run (assume data already in cache)?
  - Easy model: L1 Rate (needs 2 8-byte doubles/floating point add/multiply). If 32 GB/sec, then 4GFlops is possible with a 2GHz clock
  - But it is not that simple…



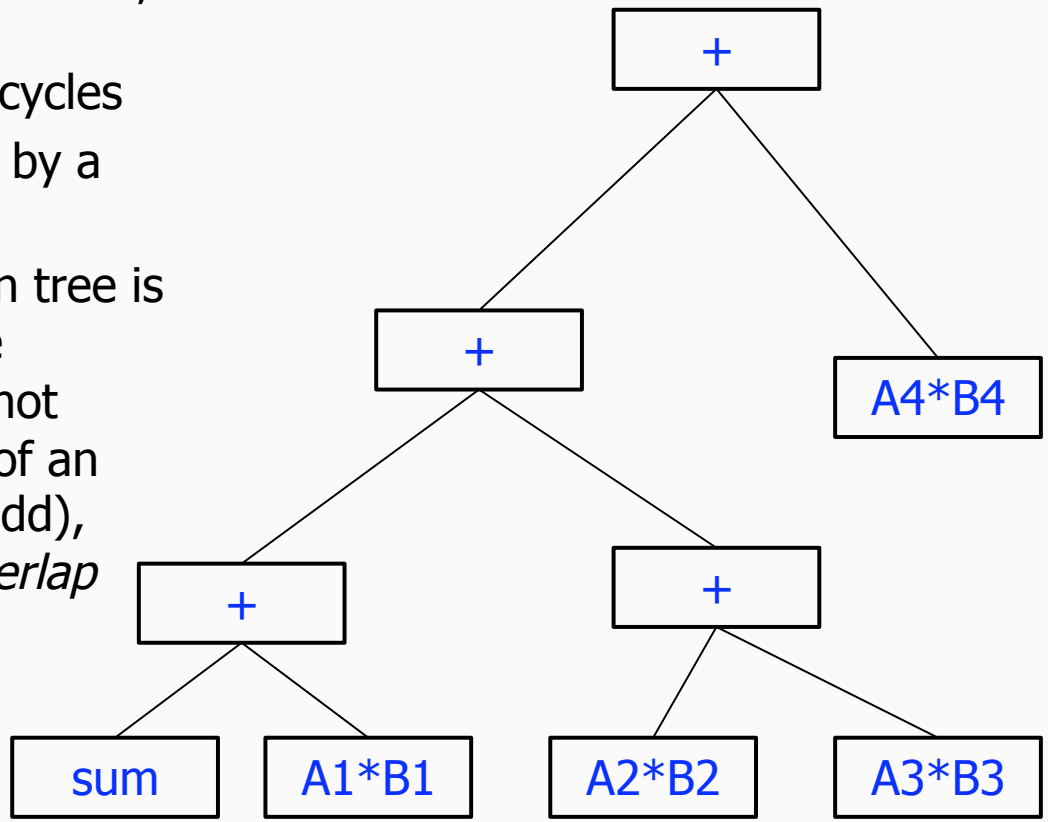
# Operation Order

- Fortran specifies the operation order. The loop defines the result as
  sum = ((((a$_1$*b$_1$) + (a$_2$*b$_2$)) + (a$_3$*b$_3$))+(a$_4$*b$_4$)) + …



# Impact of Implied Dependencies

- Assume each add takes 3 cycles to complete (latency)
- Since each add depends on the result of the prior add, only 1 add may be performed every 3 cycles
- Top speed reduced by a factor of 3
- But if the evaluation tree is balanced, there are separate adds (do not involve the results of an immediately prior add), those adds may *overlap*




# Associativity

- Operations that are associative maybe evaluated in any grouping:
- (a*b)*c = a*(b*c)
- Floating point is *not* associative
  - Round-off error, e.g., can lead to large errors.
- Options available:
  - Rewrite code to provide an order of evaluation with fewer immediate dependencies
  - Tell compiler to assume all operations are associative (*all*, in the entire file)
    - — Some higher-levels of optimization imply this
    - — Without it, latency of pipeline operations severely limits performance
    - — But with it, enabling optimization can *change the computed* results!


# A Faster Reduce

- Even better is to divide the tree. Consider this code instead
  - Performance of original version: 530 Mflops
  - Performance of new version: 1700 Mflops
- (Caveat: The new version has some other advantages)

```
sum1 = 0.0
sum2 = 0.0
sum3 = 0.0
sum4 = 0.0
do i=1,vecSize,4
    sum1 = sum1 + vecA(i)*vecB(i)
    sum2 = sum2 + vecA(i+1) * vecB(i+1)
    sum3 = sum3 + vecA(i+2) * vecB(i+2)
    sum4 = sum4 + vecA(i+3) * vecB(i+3)
enddo
sum = sum1 + sum2 + sum3 + sum4
```

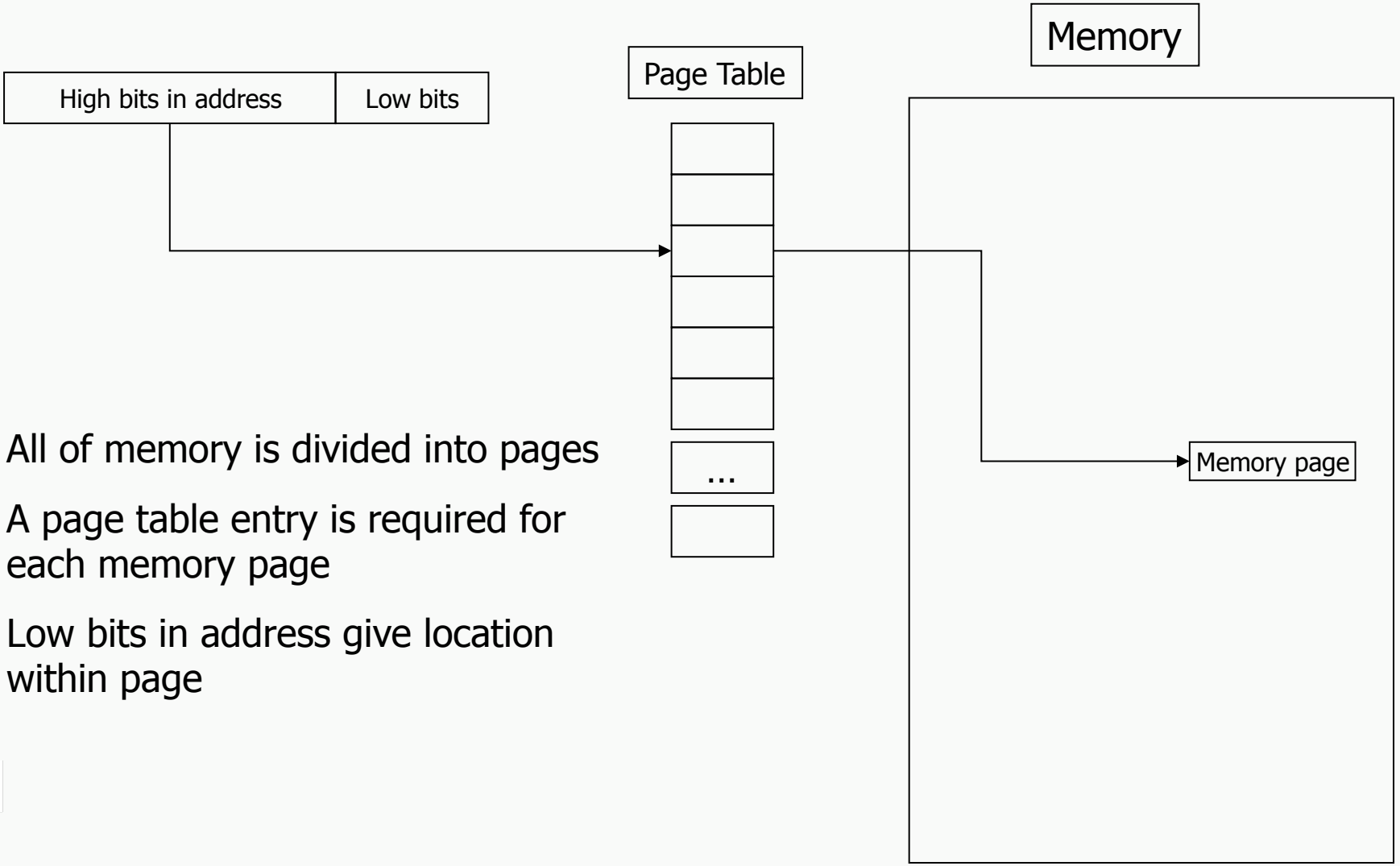

# Virtual Memory

- So far, we've assumed that the process is addressing "memory"
- In most systems, (user) processes use "virtual" addresses
  - Gives the process the illusion that it directly addresses all real memory
  - Gives the process the illusion that there is more real memory than is really available


# How Virtual Memory Works

- Memory is divided into blocks called *pages*
  - Each address has two parts
    - —Low bits: location of item within a page
    - —High bits: page number
  - Pages are mapped to different parts of the real memory *or* stored on denser (but slower) media (typically disk)

# Paging Example




All of memory is divided into pages

A page table entry is required for each memory page

Low bits in address give location within page

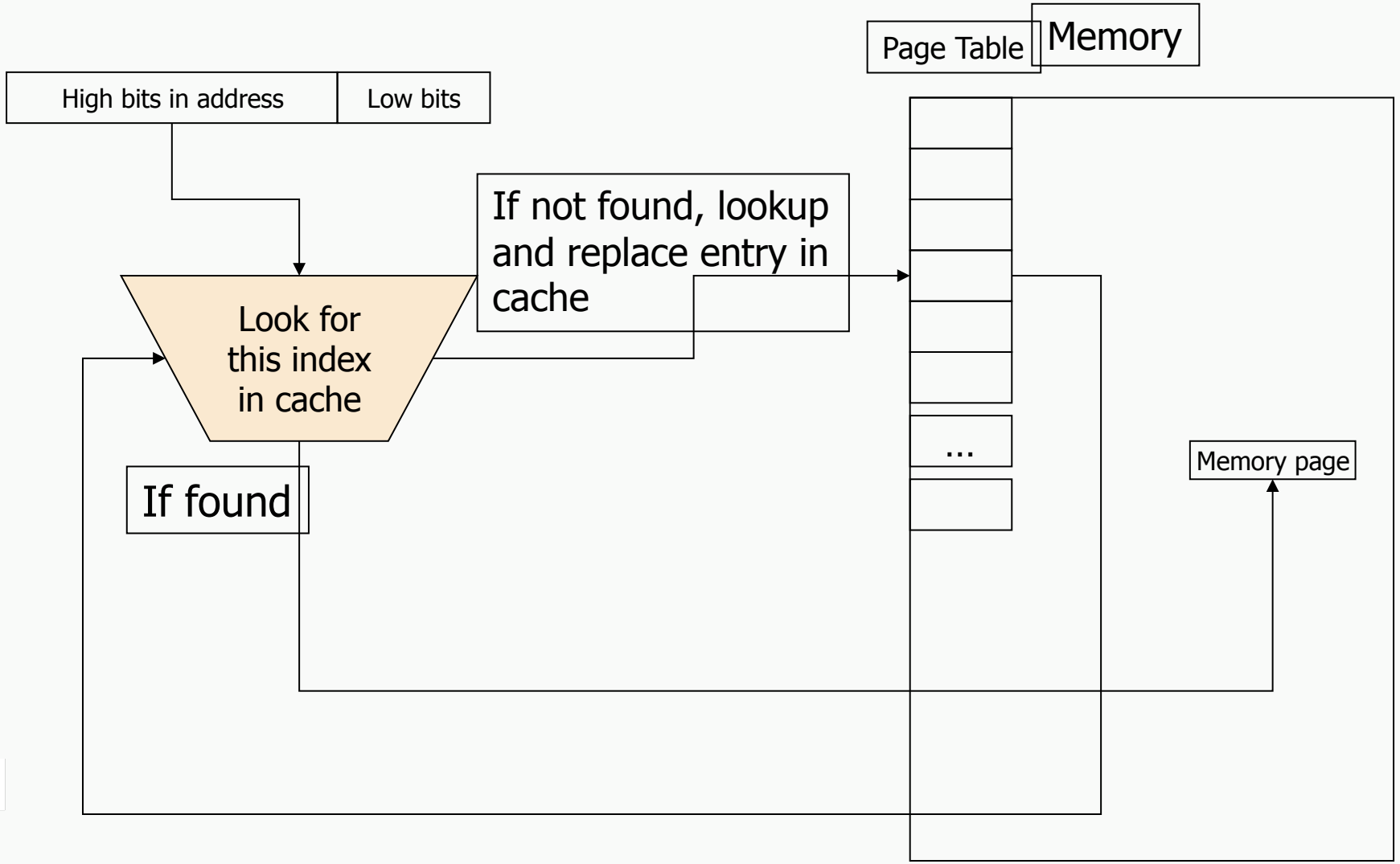

# Implementing Paging

- Virtual memory introduces some costs because the virtual address must be translated to a physical address
- Consider this case:
  - Let each page contain 4k bytes
    - —A common size
  - Address uses lower 12 bits to represent location in the page
  - Upper bits give page number
    - —For a 32-bit address space (4GB of memory), use the top 20 bits
- For each page number, there is a corresponding location
  - Either in physical (real) memory
  - On "backing store" (in the *swap* file on disk)


# Page Tables

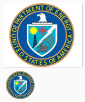
- Each page number requires a *mapping* to real storage
  - If all 4GB memory is real, require $2^{20}$ entries of 4 bytes each, or $2^{22}$ bytes of memory
    - —This is 4MB
    - —As large or larger than many cache memories
      - Using a single table to map all of virtual memory to real memory will be slow (memory lookup is latency bound)
    - —A common solution is to use a cache
      - Every address used by the program must be translated from a virtual address to a physical address
      - Must be *very* fast
      - Typically small (e.g., 64 pages (entries))

# Paging Example With Cache




# Translation Lookaside Buffer (TLB)

- The page mapping cache is called a Translation Lookaside Buffer (TLB)
  - Lookup is not easy when it has to be very fast
  - As a result, TLBs are often small but fast enough to return physical address quickly
- What happens on a page miss (entry is not in the TLB)?
  - Fetch entry from memory (the whole page table isn't big relative to main (DRAM) memory
    - —Main memory latency cost




# When Virtual Memory Exceeds Physical Memory

- Virtual memory allows the memory size (apparently) available to a process to exceed the available physical memory
  - If multiple processes are sharing memory, virtual memory allows each process to act as if it has all of the memory
- Where is the data actually stored (it has to be somewhere)?
  - Typically stored on disk in a *swapfile* (also called *secondary storage*). May be stored in other slow but high-density memory
- The page table indicates whether the location of the memory is in physical (fast) memory or if it is in secondary storage


# TLB Revisited

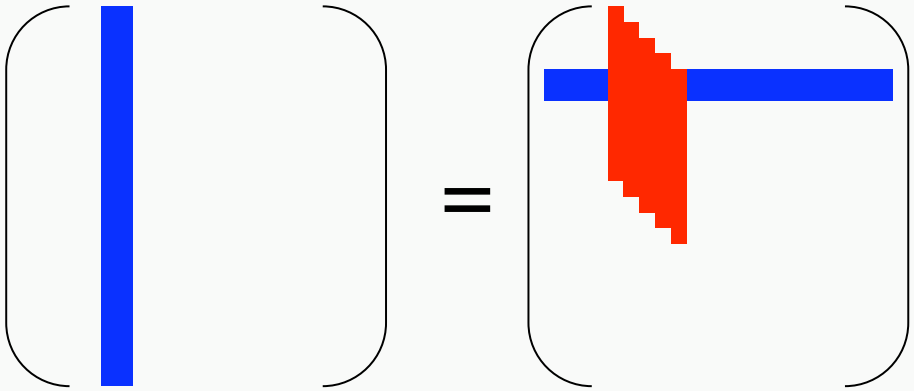
- When an page location is not found in the TLB, first find the entry in the page table
  - Requires a memory read - latencies of 20 to 100s of cycles.
- Determine if the page is stored in the main memory (resident) or has been moved to slower disk storage
  - If resident, replace a TLB entry with the location of this page and return the physical address
  - If not resident, transfer control to the operating system to handle a page fault
    - —A page fault has latencies in milliseconds (time to find and read data from disk)


# Impact on Algorithms

- Large cost if data outside of TLB set is accessed frequently
- Consider the transpose example with a 2048 x 2048 matrix and a TLB with 64 entries
- Each entry an 8-byte double precision value

# Simple Example: Matrix Transpose (From TLB Viewpoint)

- Do j=1,n
  do i=1,n
  b(i,j) = a(j,i)



- No temporal locality (data used once)
- Spatial locality only if (words/pagesize) * n fits in TLB
  - Otherwise, each column of a may cause a TLB miss


# Transpose with 4K pages:

- Each column of the matrix requires 4 pages
  - A page is mapped for stores every 512 rows
  - A page is mapped for loads on every column:
    - —Use only a single entry from a page before going to the next one
    - —Process 2k-1 pages before returning to a previous page
    - —*Every* load incurs a TLB miss


# Transpose with 64k pages

- 4 columns per page
  - It takes 512 pages to cover one row of the matrix
  - But get 4 values out of each page
    - —Every fourth load incurs a TLB miss

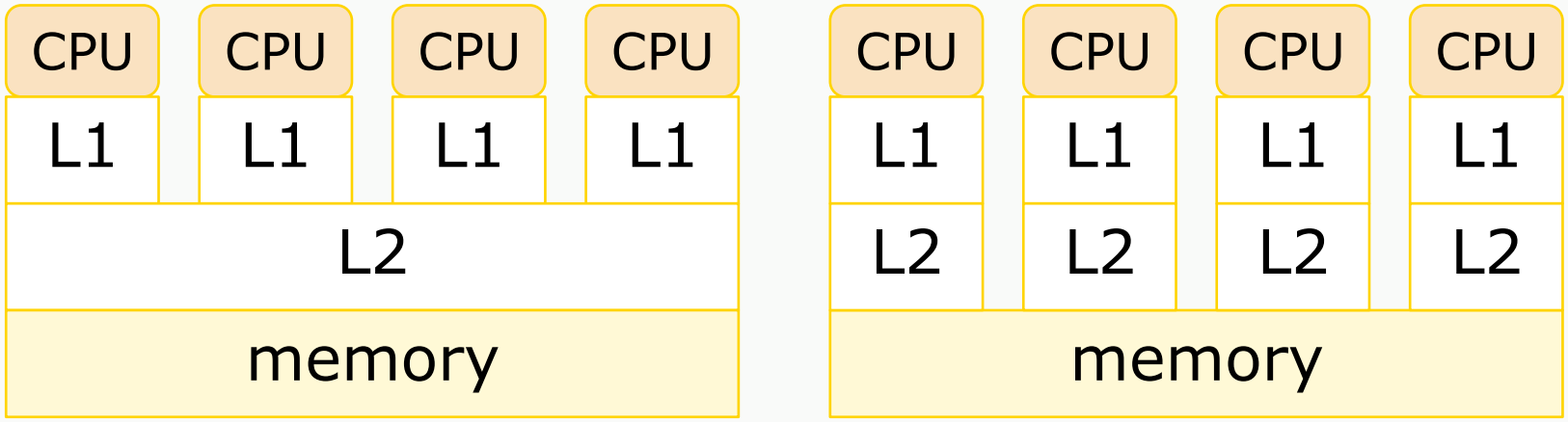

# Transpose with 1MB pages

- 64 columns per page
  - It takes 32 pages to cover a row
  - Only compulsory TLB misses
    - —Compulsory misses are the ones that cannot be avoided - they occur the first time that a page address is used


# Shared Memory Systems

- Loads and stores are supported in hardware
- Permits a (apparently) simple, low-overhead programming model: multiple threads, each executing standard assignment/reference operations
- Many correctness and performance hazards, however….

# Two Architectural Models for Shared Memory Nodes




- Left: Shared variables can stay in fast memory
  - But ensuring correctness complicates design, can impact performance
- Right: Updates to shared variables requires store to memory (slow)
  - But easier to design; faster for partitioned address space models


# Complications

- Consistency
  - When does one thread see the results of an update to memory made by another thread?
- Sequential consistency
  - Execution is as if the execution is some interleaving of the *statements* (not the hardware instructions)
  - Code then executes "the way it looks"
- Sequential consistency is hard to make fast
  - Other consistency models trade simplicity for performance
  - Release consistency requires separate ac*quire* and *release* actions on an object

# More Complications

- Writes may be completed in an order that is different than the were issued. Consider this code:

| Thread 0 | Thread 1 |
|---|---|
| A=1;<br>B=2;<br>A=0; | B=3;<br>While (A);<br>Printf( "%d\n", B ); |

What value is printed?
Does it matter if A and B are declared volatile?

If sequential consistency is provided, then the value printed is known.


# False Sharing

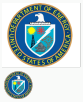
- Consider this code:

| Thread 0 | Thread 1 |
|---|---|
| **N=100000;**<br>**While (N--) a++;** | **M = 100000;**<br>**While (M--) b++;** |

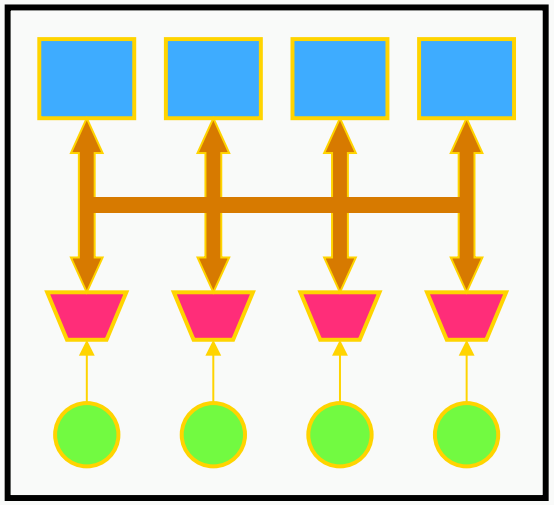
How many cache misses occur?
1 Model:
There are 4: 1 each for the variables N, M, A, B.


# False Sharing (2)

- Consider this case
  - A, B, N, M are all in the same cache line
  - A processor may only write to a value if it is in that cores L1 cache
  - A and B are written to memory (store), not just updated in register
- Then instead of 4 cache misses, there are as many as 200000 (one for each access to either A or B)
- This is not a correctness problem; it is a performance problem
  - The programming language *hides* the hardware-defined associating between variables



# Symmetric Multiprocessor (SMP)



Symmetric Multiprocessor

Memory =

Arithmetic/Logic =

Control Logic =

Communications =

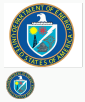
Control Signals =


# Distributed Shared Memory (DSM)

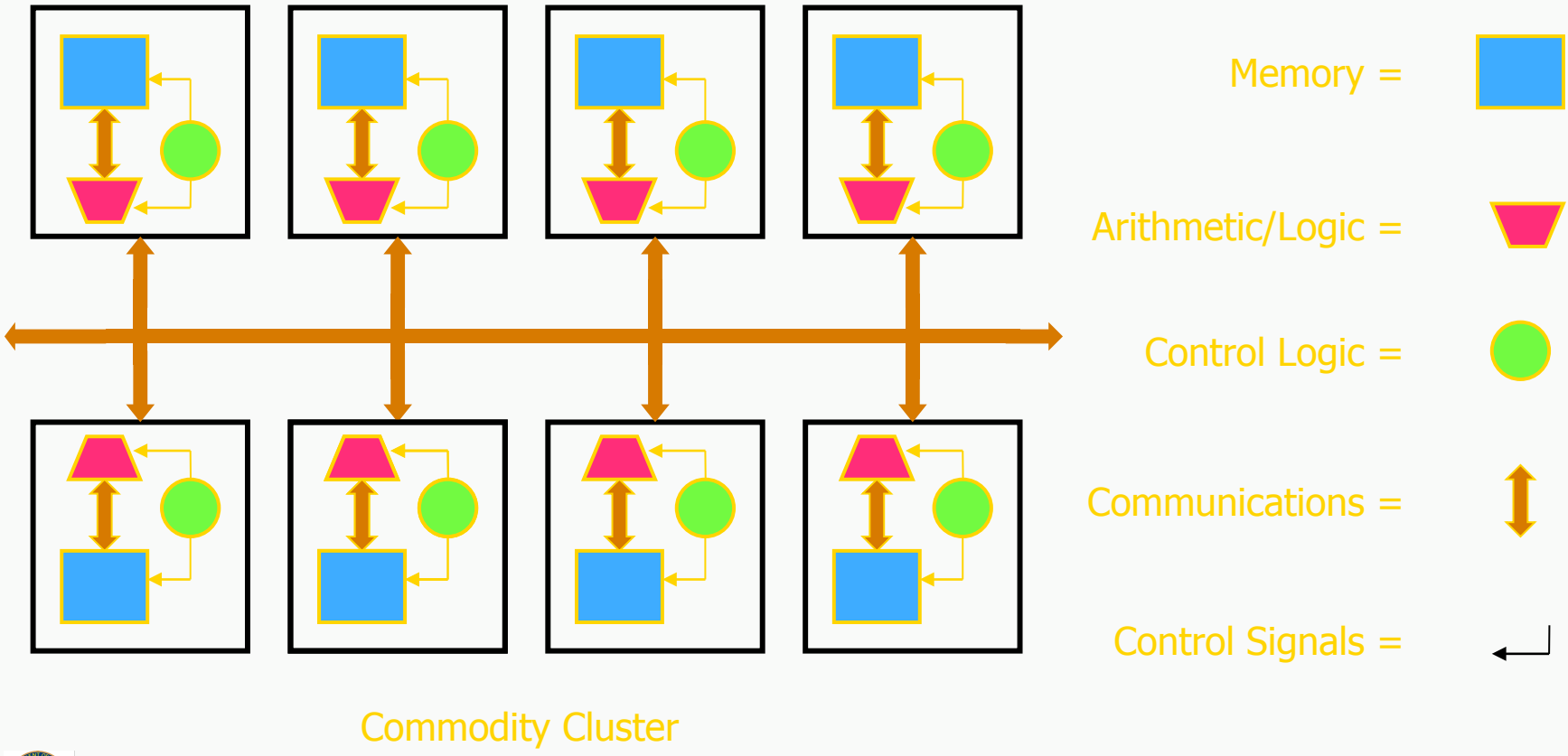
- Multiple microprocessors share a single global memory
  - Any processor can access any global memory bank
  - Shared interconnect between multiple processors and multiple memory banks
- Access time to memory from all processors is different
  - NUMA – uniform memory access
  - Depends on distance or number of hops through network
- Cache coherent
  - Any modification to global data by one processor is reflected by the caches of the other processors
  - Guaranteed sequential consistency
- Programming model multiple threads with shared memory
  - e.g., OpenMP
- Scaling demonstrated to the range of a thousand processors


# Commodity Clusters

- Multiple stand-alone computers (nodes) working together
  - Compute nodes are commercially available from vendors
  - e.g., personal computers, workstations, servers
- Interconnected by one or more private (unworldly) networks
  - Network is not visible from the external environment
  - Network is commercially available from vendors
  - e.g., Ethernet, Myricom Myrinet, Quadrics QSW, Infiniband
- Nodes are single processor computers or SMPs
  - Processors within each node may share all of global memory
- Nodes do not directly access each other's memory
  - Distributed memory systems
  - Data is transferred between nodes by I/O (network) through software
- No Cache coherence between nodes
  - All data sharing is managed by programmer and software between nodes
- Programming model is communicating sequential processes (CSP)
  - e.g., MPI
- Single largest class on the Top-500 List

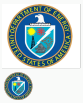


# Commodity Cluster




# Distributed Memory Massively Parallel Processor

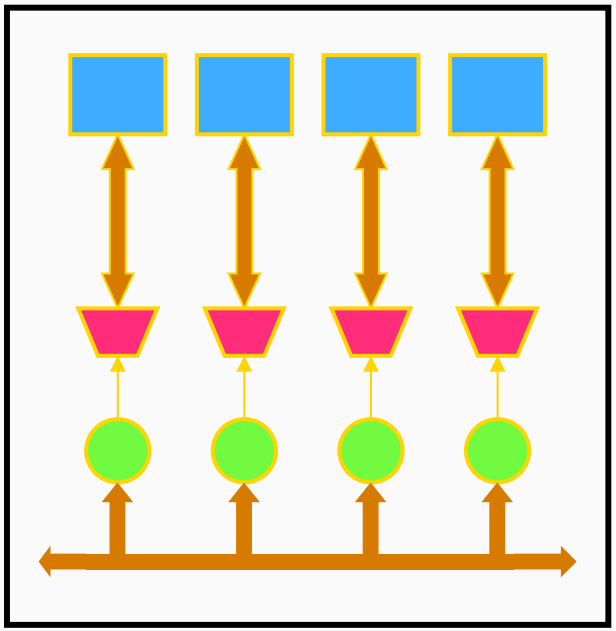
- Much like clusters, except
  - High-performance interconnect
    - —May be organized around memory transfers rather than network packets
  - Scale
    - —Tens to hundreds of thousands of processors now
- Same programming model
  - May have integrated hardware support for the programming model
  - Integrated systems management




# Distributed Memory MPP

- A tightly coupled ensemble of (possibly modified) commodity microprocessors and DRAM memory banks
- Interconnection of elements by means of custom high bandwidth network
  - Cross-bar switch
  - Omega / banyan
  - Mesh / toroidal
  - Clos / "Fat-tree"
- Each node comprises
  - A microprocessor (sometimes more than one)
  - A block of main memory (possibly multiple banks)
  - High bandwidth interface to system network(s)
  - Possibly I/O channels to secondary storage and external environment
- Node memory name space segregated from each other
- Programming model mostly MPI

# Distributed Memory Massively Parallel Processor (MPP)



Distributed Memory
Massively Parallel
Multiprocessor

Memory =

Arithmetic/Logic =

Control Logic =

Communications =

Control Signals =


# Hardware Support for Parallel Programming

- Distributed Memory Parallel systems support several models of interaction between the nodes or processors
  - Two-sided communication
    - —Often described as send-receive, there are separate actions for sending and receiving data
    - —The network hardware *may* provide extra support for this, or it may be simply provide basic operations to move data and signal that data has arrived.
  - One-sided communication
    - —Often described as put/get or remote load/store
    - —The network hardware may provide direct implementation of this, particularly if virtual memory is not used and absolute addresses can be used.


# I/O and Parallel I/O

- Can be viewed as another memory hierarchy
  - Latencies are *much* higher
- Memory consistency issues
  - Like shared memory, providing memory consistency trades performance for ease of use
  - POSIX I/O provides strong consistency…
  - But few file systems provide both POSIX consistency and performance


# Common Themes

- Understanding memory hierarchies is critical to *designing* good algorithms, not just writing fast code
- The order (or lack of order) of operations can have a major effect on performance
- Parallelism exists even within a single "core" of a processor


# What's Next

- How do we express algorithms to exploit different computer architectures?
  - Through parallel programming models…