Parallel Programming

Lecture 02: Overview of Parallel Architectures and Programming Model

谭光明

Outline

- Parallelism
- Overview of parallel machines (~hardware) and programming models (~software)
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs or GPUs
  - Grid/Cloud

Parallelism and Concurrency

Concurrency

"Concurrency occurs when two or more execution flows are able to run simultaneously."

A property of execution flows

Parallelism

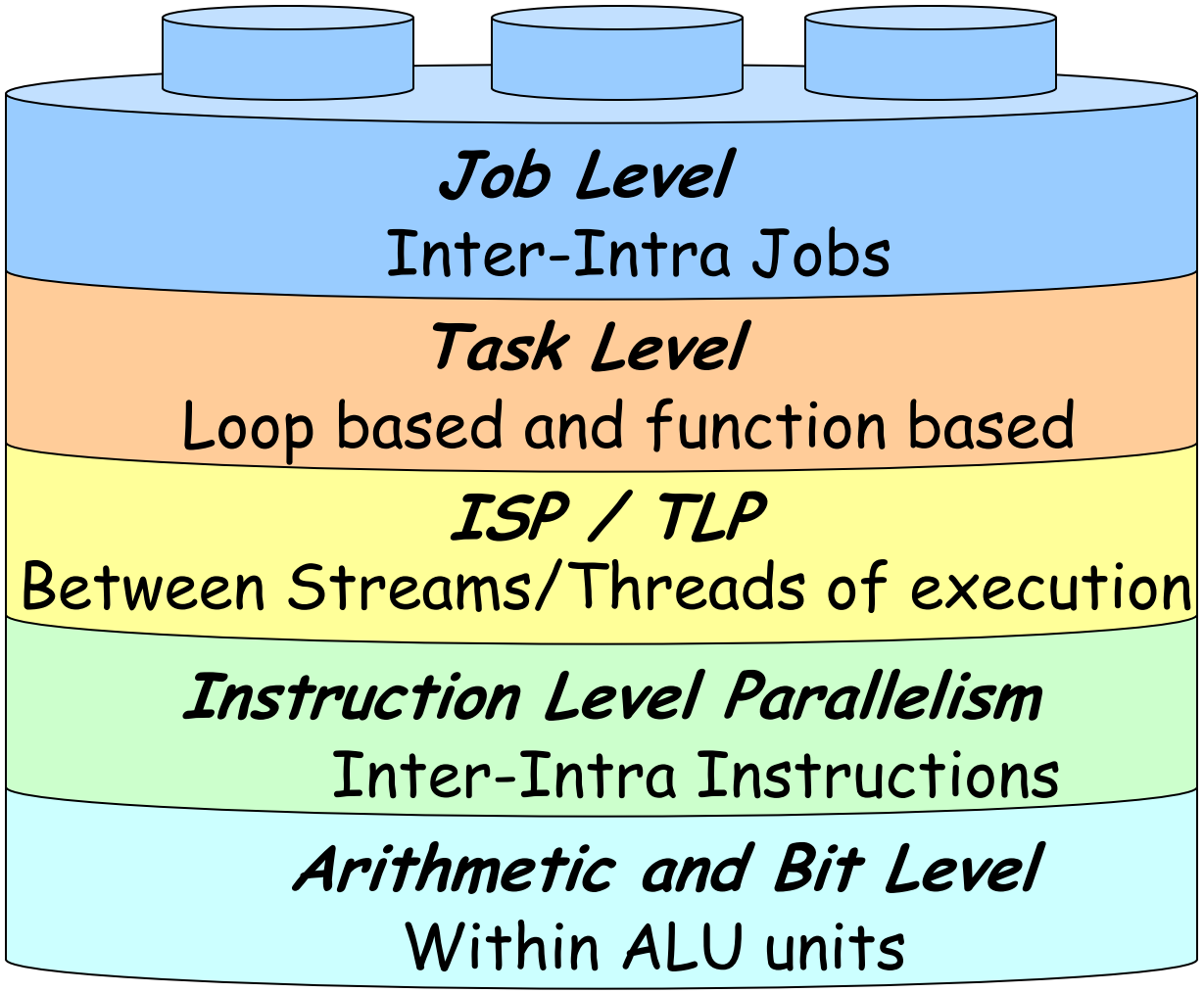
"The maximum number of independent subtasks in a given task at a given point in its execution."

A state of execution flows

Parallel Computing ➔ Division of tasks in Parallel Computers and side effects (i.e. Memory consistency)

Concurrent Computing ➔ Interactions between tasks in a concurrent system (i.e. Signal Handling)

Type of Parallelism



Job Level Parallelism

Orchestrator: **OS, Programmer**

Overlap of Jobs with other CPU activities.
Example: I/O retrieval

Inter-Job Parallelism:

Switch between jobs when "processing" lengthy I/O request.

	Time 1	Time 2
CPU	Job 1	Job 2
I/O	Job 2	Job 1

Intra-Job Parallelism:

Compute an independent calculation while waiting for a DMA transfer.

Requirements: **Duplicated resources.**

Task Level Parallelism

Orchestrator: Compiler, Programmer, OS

Task Level == Program Level

Among Different Code sections

Functions calls or other abstractions of code (like code blocks)

Different Iteration of the same Loop

Data Dependency and Program Partitioning

Thread Level Parallelism

Thread

A sequence of instructions which has a single PC (program counter).

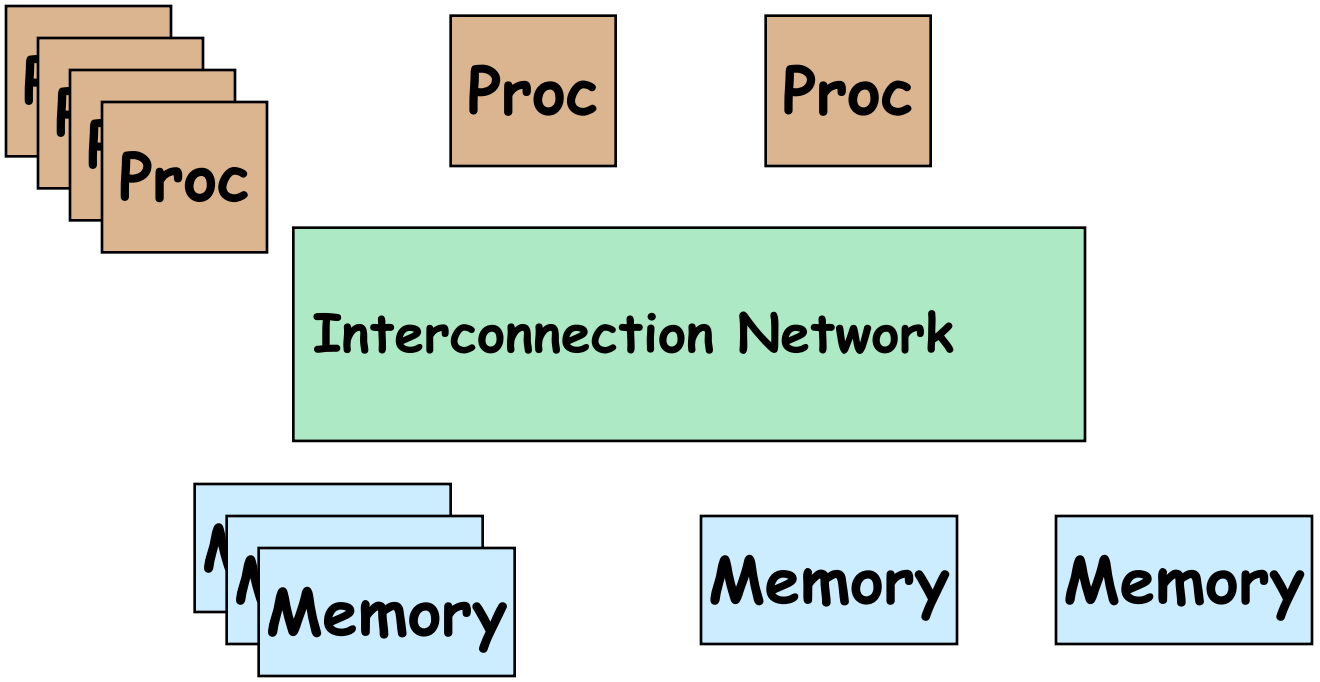
Designed to run different parts of a program in different processors to take full advantage of parallelism

Orchestrator: Programmer or Compiler

Instruction Level Parallelism

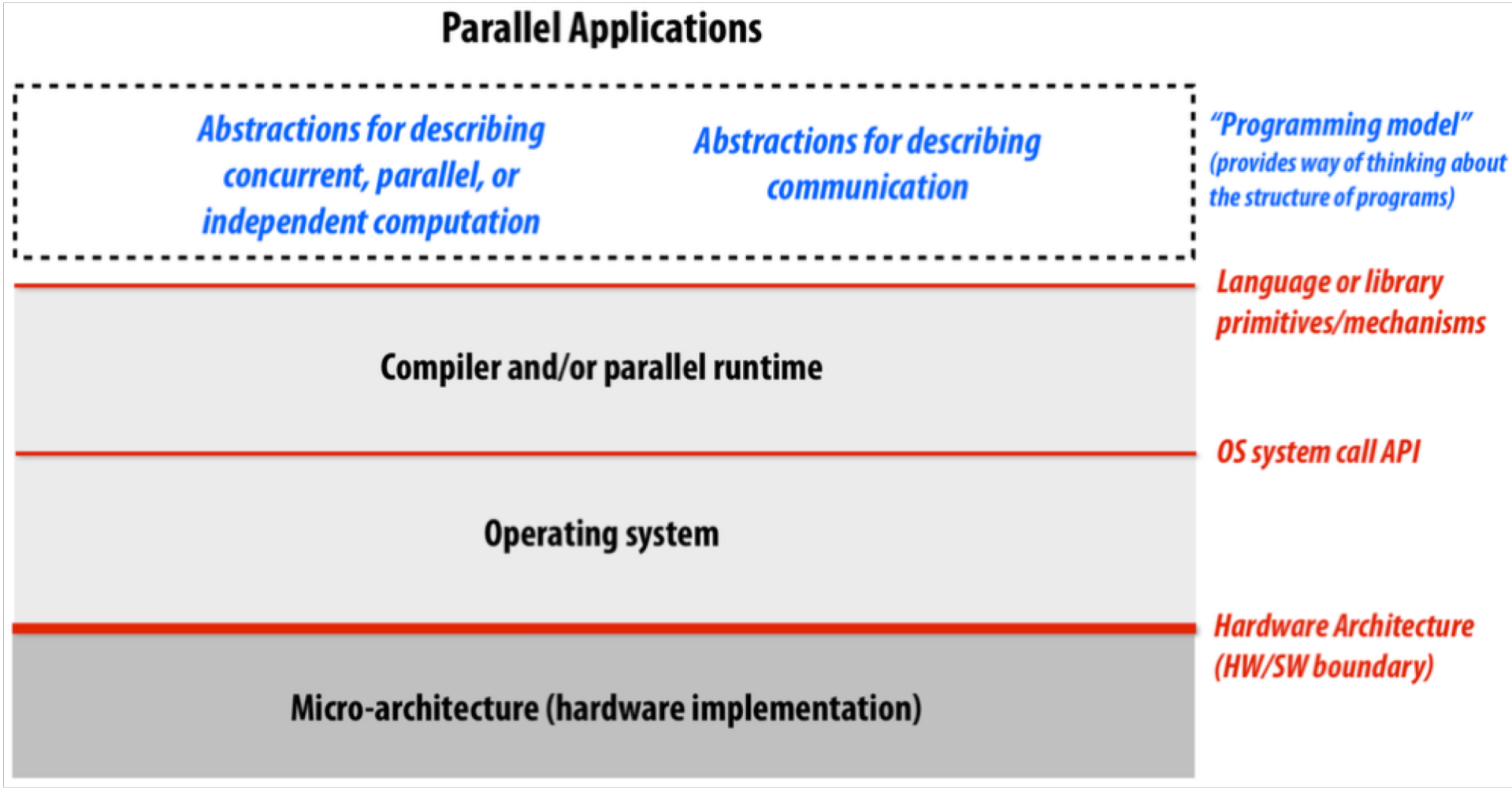
- **Between Instructions**
  - Independent Instructions running on hardware resources
  - Assumption: There is more than one hardware resource (ALU, Adder, Multiplier, Loader, MAC unit, etc)
  - Data Dependency
- **Between Phases of Instructions:**
  - Examples: Instruction Pipelines

A Generic Parallel Architecture

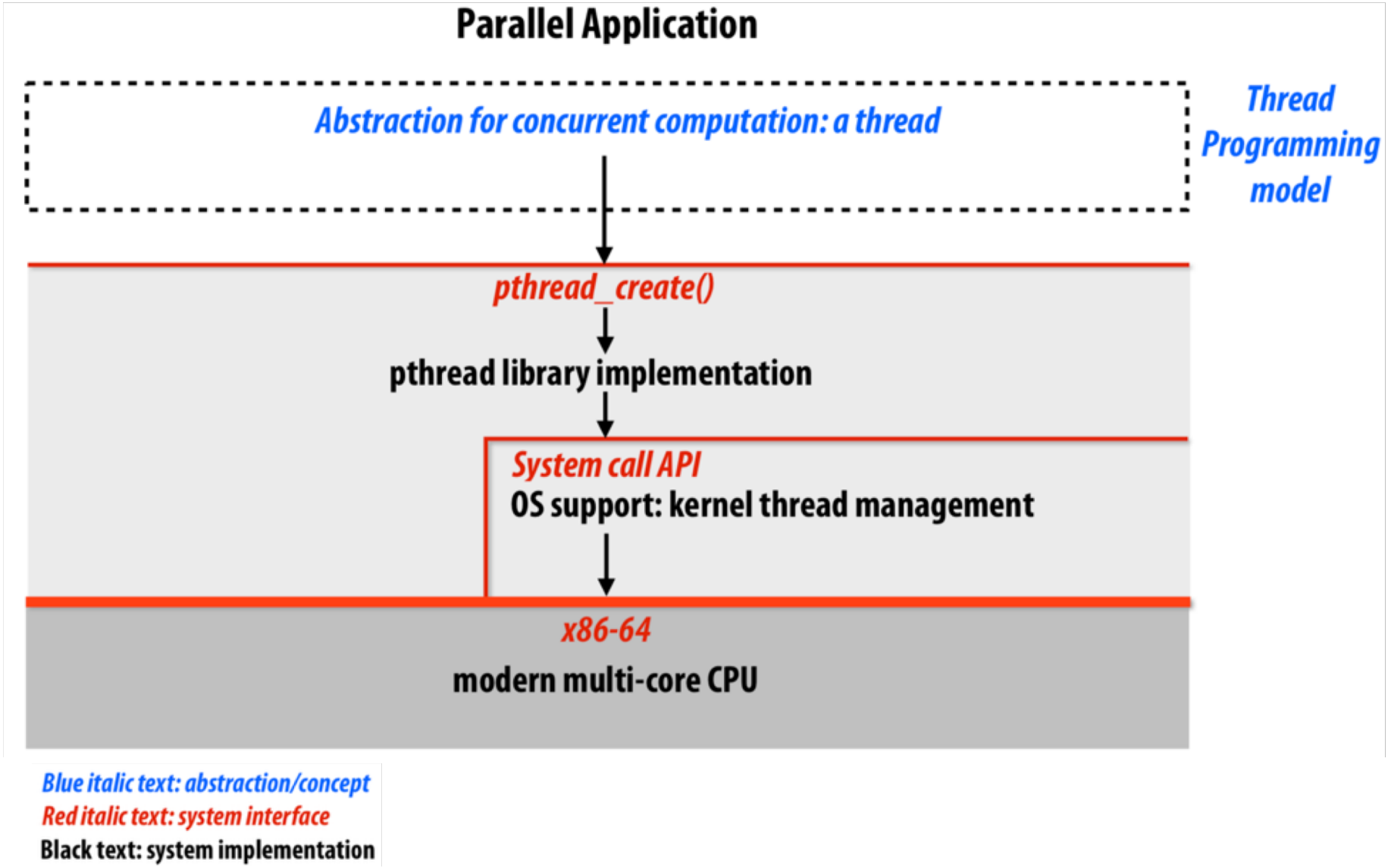


- Where is the memory physically located?
- Is it connected directly to processors?
- What is the connectivity of the network?

Abstraction vs. Implementation



Example


Parallel Application
Abstraction for concurrent computation: a thread
Thread Programming model
pthread_create()
pthread library implementation
System call API
OS support: kernel thread management
x86-64
modern multi-core CPU
Blue italic text: abstraction/concept
Red italic text: system interface
Black text: system implementation

Parallel Programming Models

- **Programming model is made up of the languages and libraries that create an abstract view of the machine**
- **Control**
  - How is parallelism created?
  - What orderings exist between operations?
- **Data**
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- **Synchronization**
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
- **Cost**
  - How do we account for the cost of each of the above?

Overview

Programming Models	Machine Models
1. Shared Memory	1a. Shared Memory 1b. Multithreaded Procs. 1c. Distributed Shared Mem.
2. Message Passing	2a. Distributed Memory 2b. Internet & Grid Computing
2a. Global Address Space	2c. Global Address Space
3. Data Parallel	3a. SIMD 3b. Vector
4. Hybrid	4. Hybrid

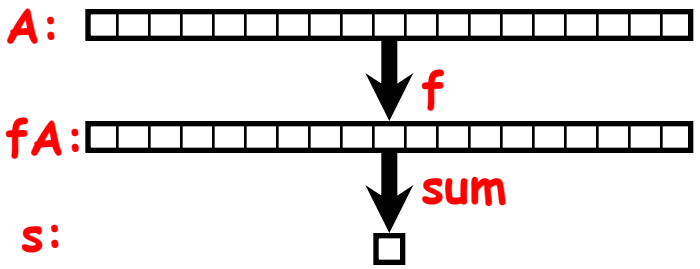
Simple Example

- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

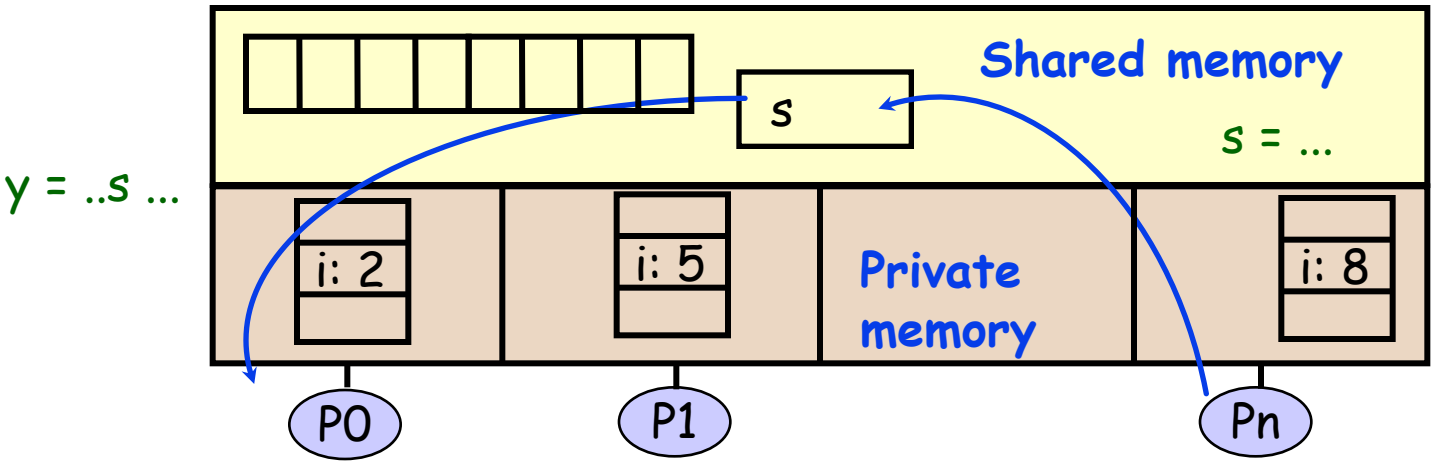
- Questions:
  - Where does A live? All in single memory? Partitioned?
  - What work will be done by each processors?
  - They need to coordinate to get a single result, how?

A = array of all data
fA = f(A)
s = sum(fA)



Programming Model 1: Shared Memory

- **Program is a collection of threads of control.**
  - Can be created dynamically, mid-execution, in some languages
- **Each thread has a set of private variables, e.g., local stack variables**
- **Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.**
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

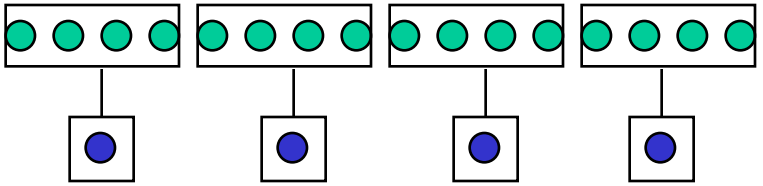


Simple Examples

- Shared memory strategy:
  - small number p << n=size(A) processors
  - attached to single memory

$$\sum_{i=0}^{n-1} f(A[i])$$

- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
  - Each computes independent "private" results and partial sum.
  - Collect the p partial sums and compute a global sum.



Two Classes of Data:

- Logically Shared
  - The original n numbers, the global sum.
- Logically Private
  - The individual function evaluations.
  - What about the individual partial sums?

Shared Memory "Code" for Computing a Sum

```
fork(sum,a[0:n/2-1]);
sum(a[n/2,n-1]);
```

```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

- What is the problem with this program?

- A race condition or data race occurs when:
  - Two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Shared Memory "Code" for Computing a Sum

A= | 3 | 5 |

f (x) = x^2

```
static int s = 0;
```

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0	9	reg1 = reg1 + reg0	25
s = reg1	9	s = reg1	25
…		…	

- Assume A = [3,5], f(x) = x^2, and s=0 initially
- For this program to work, s should be $3^2 + 5^2 = 34$ at the end
  - but it may be 34,9, or 25
- The *atomic* operations are reads and writes
  - Never see ½ of one number, but += operation is *not* atomic
  - All computations happen in (private) registers

Improved Code for Computing a Sum

```
static int s = 0;
static lock lk;
```

Why not do lock Inside loop?

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);
```

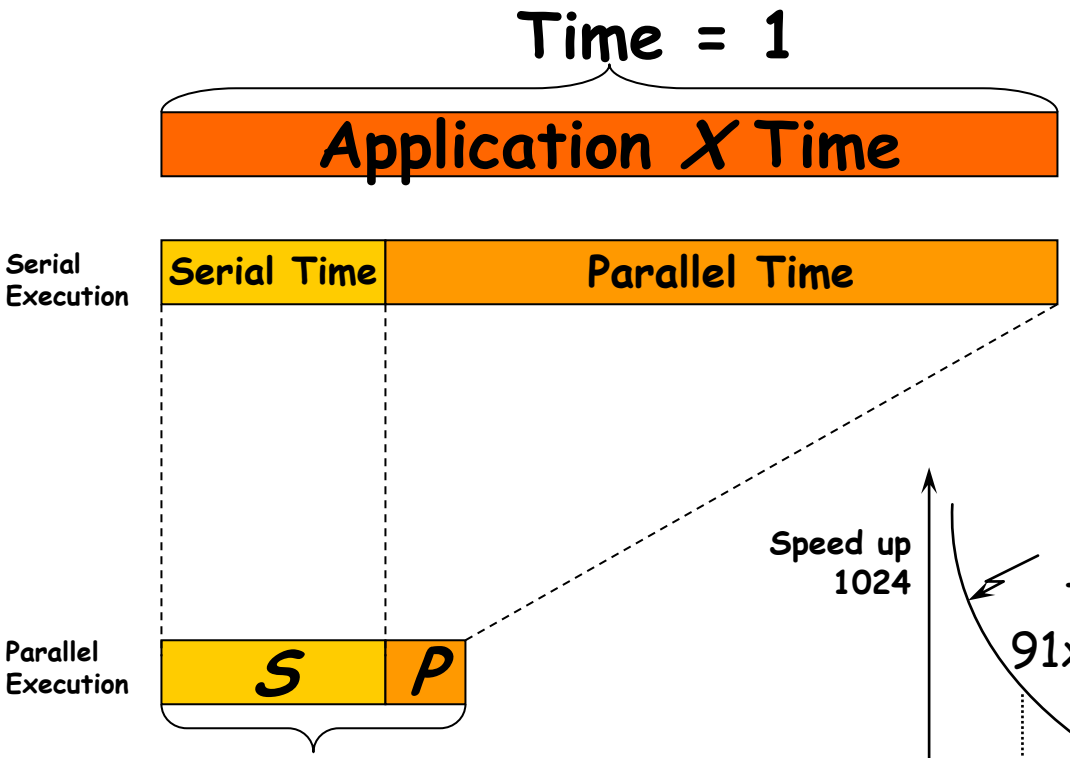
Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + f(A[i])
lock(lk);
s = s +local_s2
unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

Amdah's Law

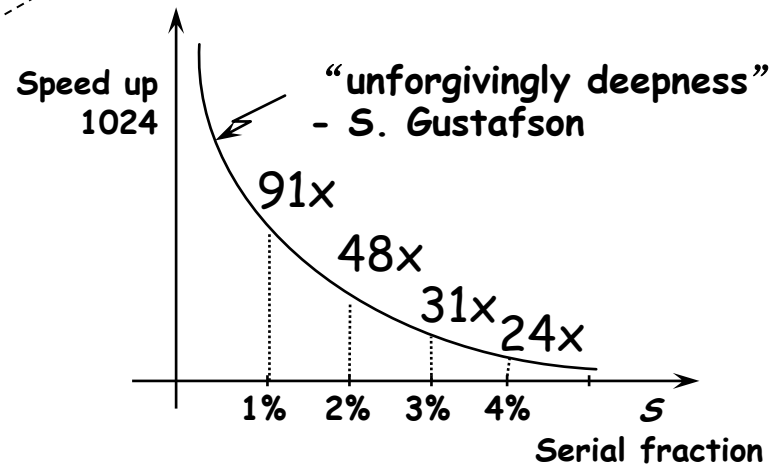
"When the fraction of serial work in a given problem is small, say *s*, the maximum speedup obtainable (from an even infinite number of processors) is only *1/s*."



$S + P = 1$

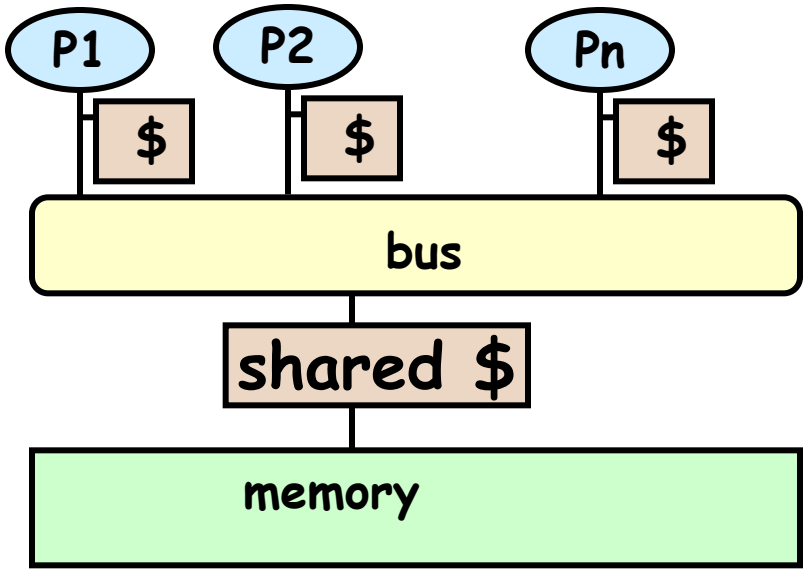
Speed Up

$$\frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}}$$



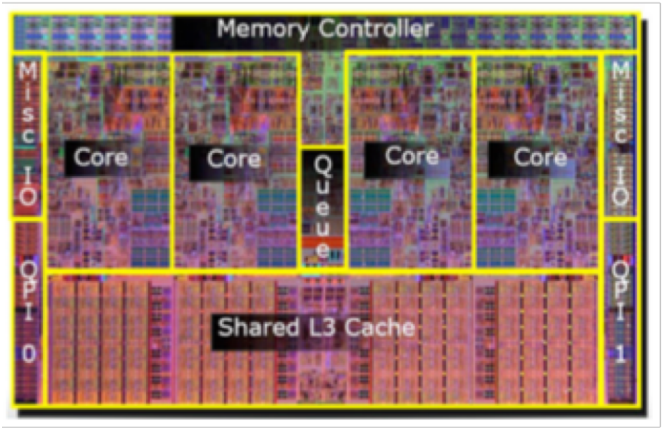
Machine Model 1a: Shared Memory

- Processors all connected to a large shared memory.
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun, HP, Intel, IBM SMPs
  - Multicore chips, except that all caches are shared
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical

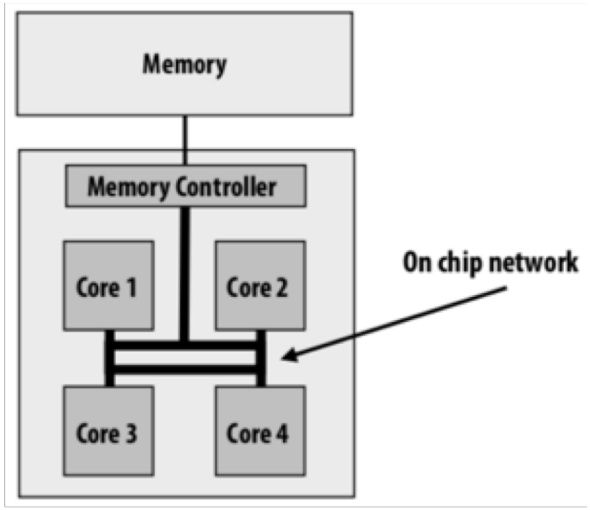


Note: $ = cache

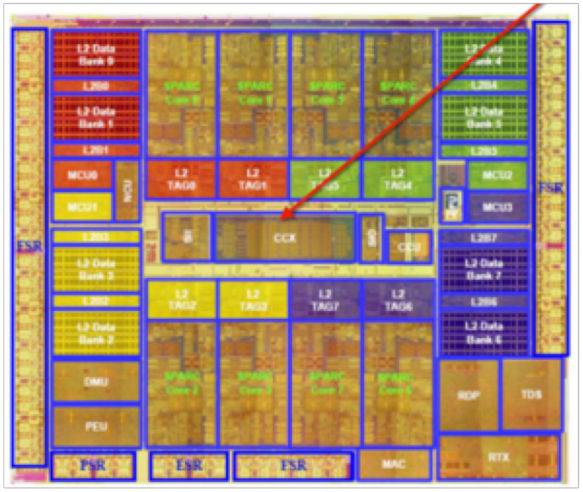
SMP Hardware



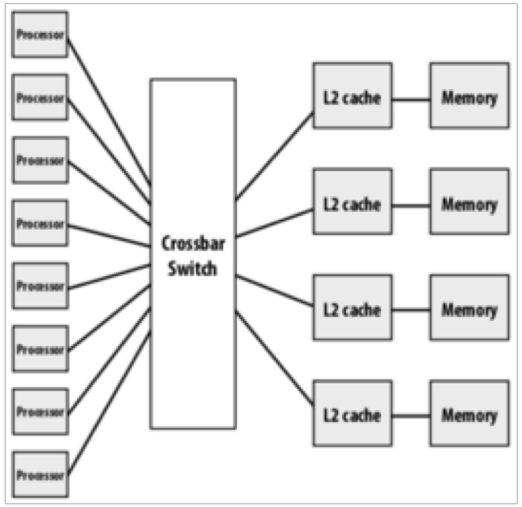
Intel



about die area of one core

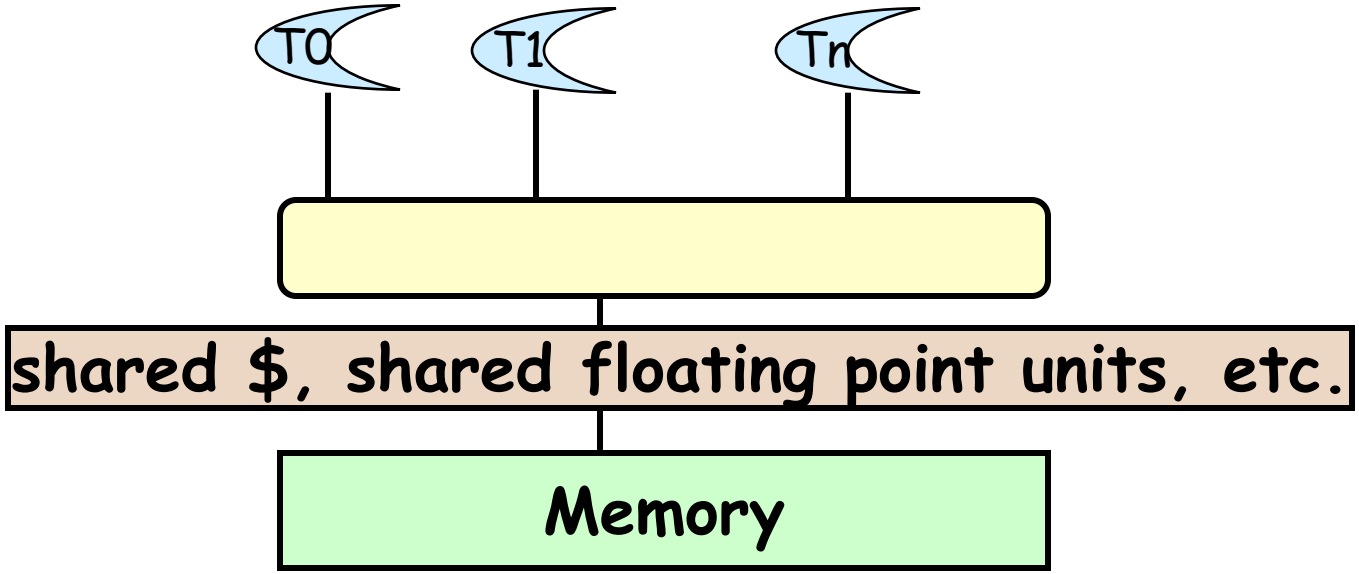


Sun Niagara2



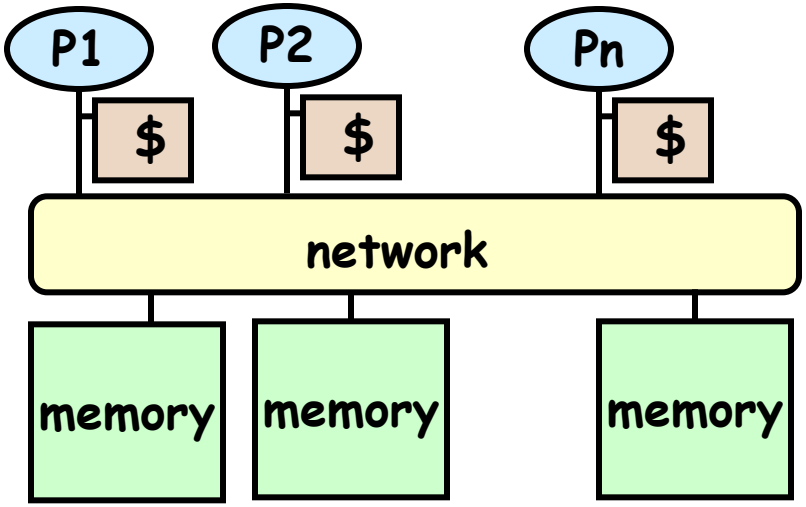
Machine Model 1b: Multithreaded Processor

- Multiple thread "contexts" without full processors
- Memory and some other state is shared
- Sun Niagra processor (for servers)
  - Up to 64 threads all running simultaneously (8 threads x 8 cores)
  - In addition to sharing memory, they share floating point units
  - Why? Switch between threads for long-latency memory operations
- Cray MTA and Eldorado processors (for HPC)



Machine Model 1c: Distributed Shared Memory

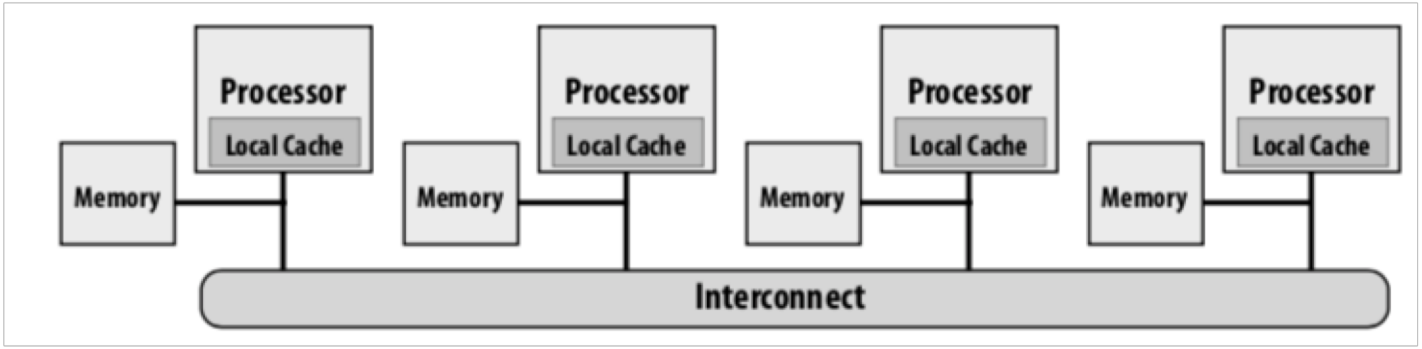
- **Memory is logically shared, but physically distributed**
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- **SGI is canonical example (+ research machines)**
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is *cache coherency protocols* – how to keep cached copies of the same address consistent



Cache lines (pages) must be large to amortize overhead
➔
locality still critical to performance

Non-uniform memory access (NUMA)

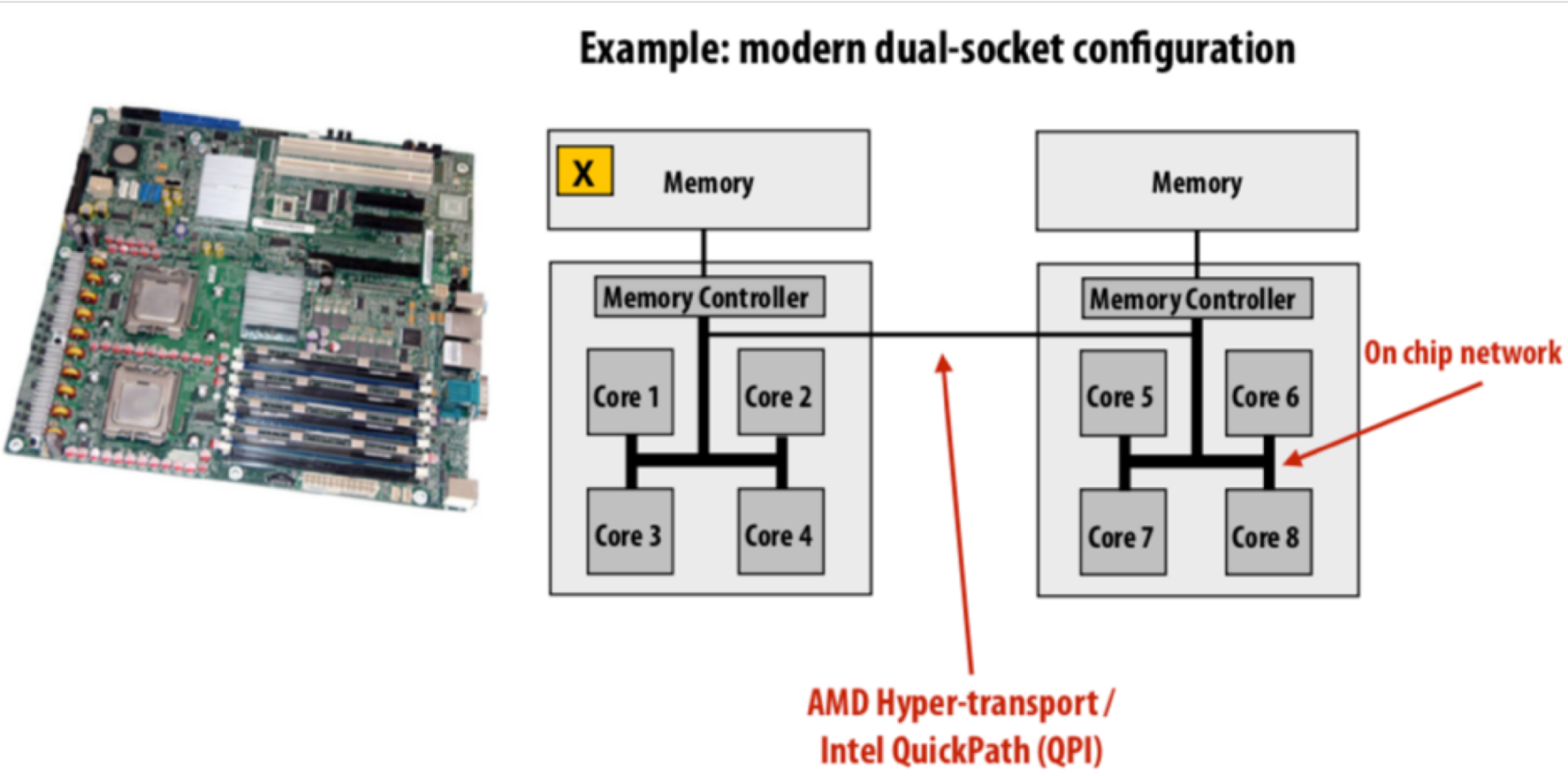
- All processors can access any memory location, but…the cost of memory access (latency & bandwidth) is different for different processors



  - Problem with preserving uniform access time in a system: scalability
    - GOOD: costs are uniform; BAD: they are uniformly bad (memory is uniform far away
  - NUMA designs are more scalable
    - Low latency and high bandwidth to local meomry
  - Cost is increased programmer effort for performance tuning
    - Finding, exploiting locality is important to performance (want most memory addresses to be to local memories

NUMA Hardware

- Example: latency to access address X is higher from cores 5–8 than cores 1–4

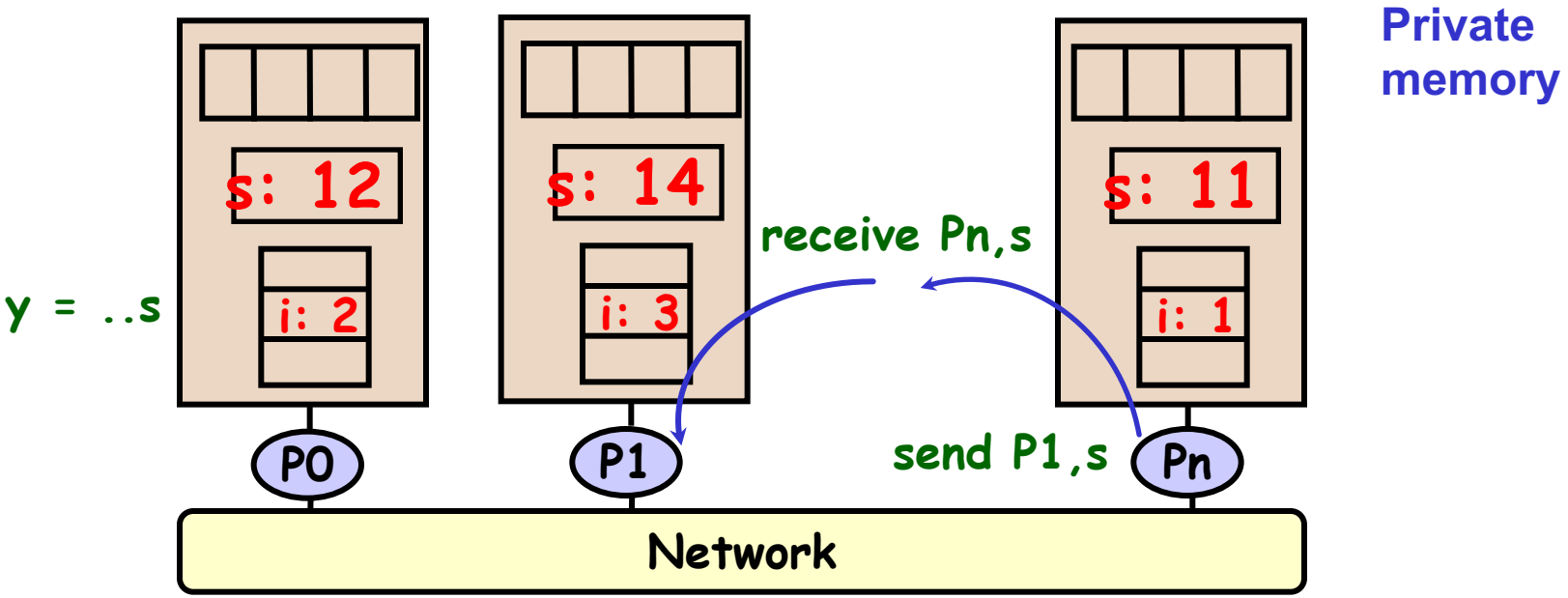


Overview

Programming Models	Machine Models
1. Shared Memory	1a. Shared Memory 1b. Multithreaded Procs. 1c. Distributed Shared Mem.
2. Message Passing	2a. Distributed Memory 2b. Internet & Grid Computing
2a. Global Address Space	2c. Global Address Space
3. Data Parallel	3a. SIMD 3b. Vector
4. Hybrid	4. Hybrid

Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW



Computing s = f(A[1])+f(A[2]) on each processor

- First possible solution – what could go wrong?

```
Processor 1
    xlocal = f(A[1])
    send xlocal, proc2
    receive xremote, proc2
    s = xlocal + xremote
```

```
Processor 2
    xlocal = f(A[2])
    send xlocal, proc1
    receive xremote, proc1
    s = xlocal + xremote
```

- If send/receive acts like the telephone system? The post office?
- Second possible solution

```
Processor 1
    xlocal = f(A[1])
    send xlocal, proc2
    receive xremote, proc2
    s = xlocal + xremote
```

```
Processor 2
    xlocal = f(A[2])
    receive xremote, proc1
    send xlocal, proc1
    s = xlocal + xremote
```

- What if there are more than 2 processors?

MPI – the de facto standard

MPI has become the de facto standard for parallel computing using message passing

Pros and Cons of standards

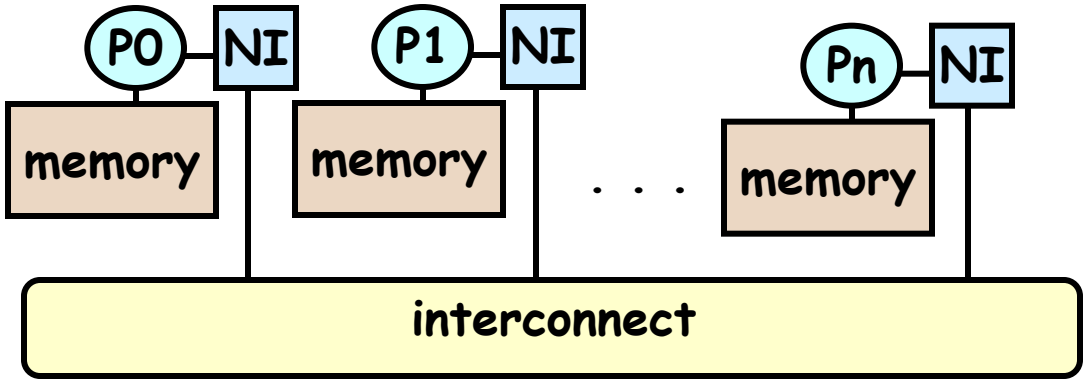
- MPI created finally a standard for applications development in the HPC community → portability
- The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation

Programming Model reflects hardware!

"I am not sure how I will program a Petaflops computer, but I am sure that I will need MPI somewhere" – HDS 2001

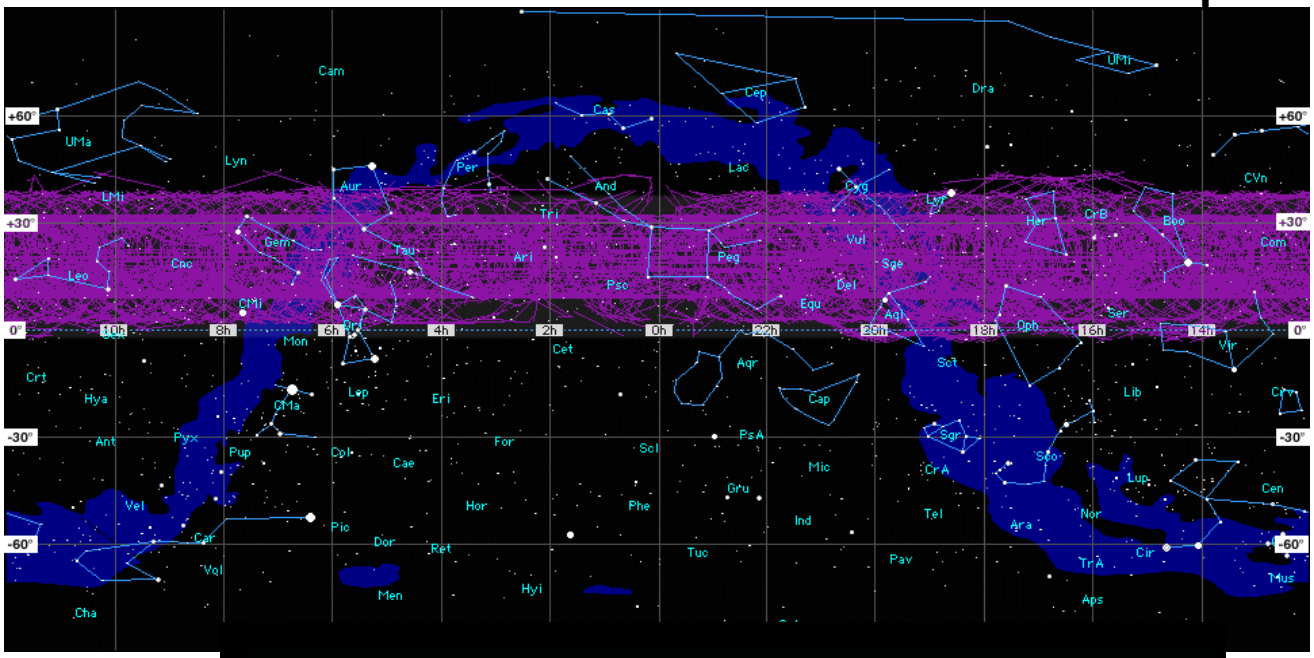
Machine Model 2a: Distributed Memory

- PC Clusters (Berkeley NOW, Beowulf)
- Most of the Top500, are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Machine Model 2b: Internet/Grid Computing

- SETI@Home: Running on 3.3M hosts, 1.3M users (1/2013)
  - ~1000 CPU Years per Day (older data)
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope

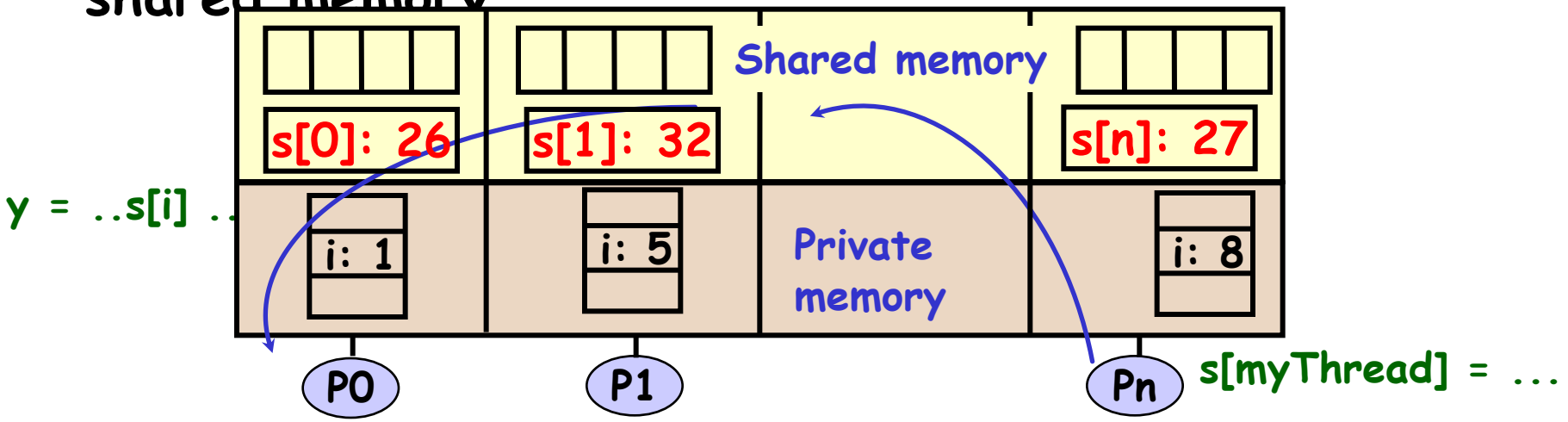


Next Step-
Allen Telescope Array

Google
"volunteer computing"
or "BOINC"

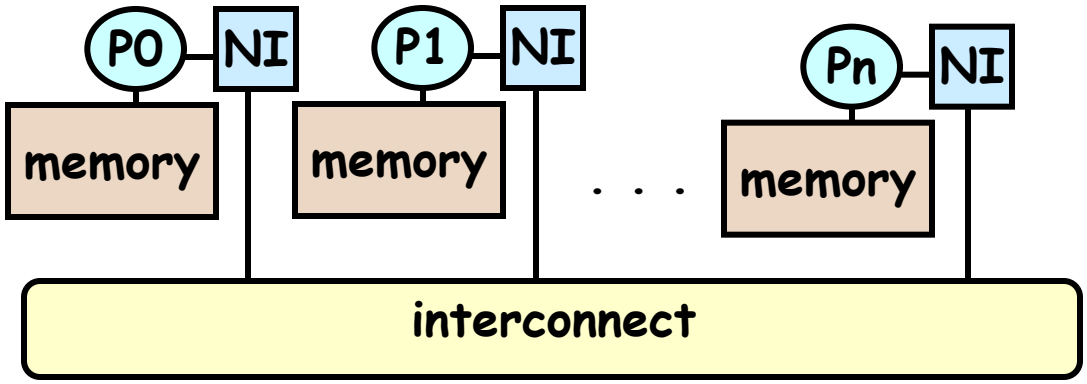
Programming Model 2a: Global Address Space

- Program consists of a collection of named threads.
  - Usually fixed at program startup time
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes
  - Cost models says remote data is expensive
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory



Machine Model 2c: Global Address Space

- Clusters built with Quadrics, Myrinet, or Infiniband
- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally



Global address space may be supported in varying degrees

Fuzzy Things

- Common to implement message passing abstraction on machines that implement a shared address space in hardware
  - "Sending message" = copying data to message library buffers
  - "Receiving message" = copying data from message library buffers
- Can implement shared address space abstraction on machines that do not support in HW (via less efficient SW solution)
  - Mark all pages with shared variables as invalid
  - Page-fault handler issues appropriate network requests
- Keep in mind: what is the programming model (abstractions used to specify program) and what is the HW implementation?

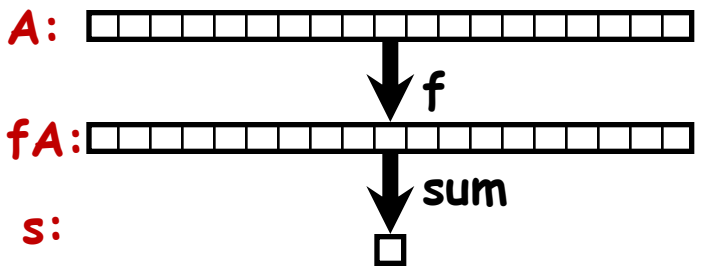
Overview

Programming Models	Machine Models
1. Shared Memory	1a. Shared Memory 1b. Multithreaded Procs. 1c. Distributed Shared Mem.
2. Message Passing	2a. Distributed Memory 2b. Internet & Grid Computing
2a. Global Address Space	2c. Global Address Space
3. Data Parallel	3a. SIMD 3b. Vector
4. Hybrid	4. Hybrid

Programming Model 3: Data Parallel

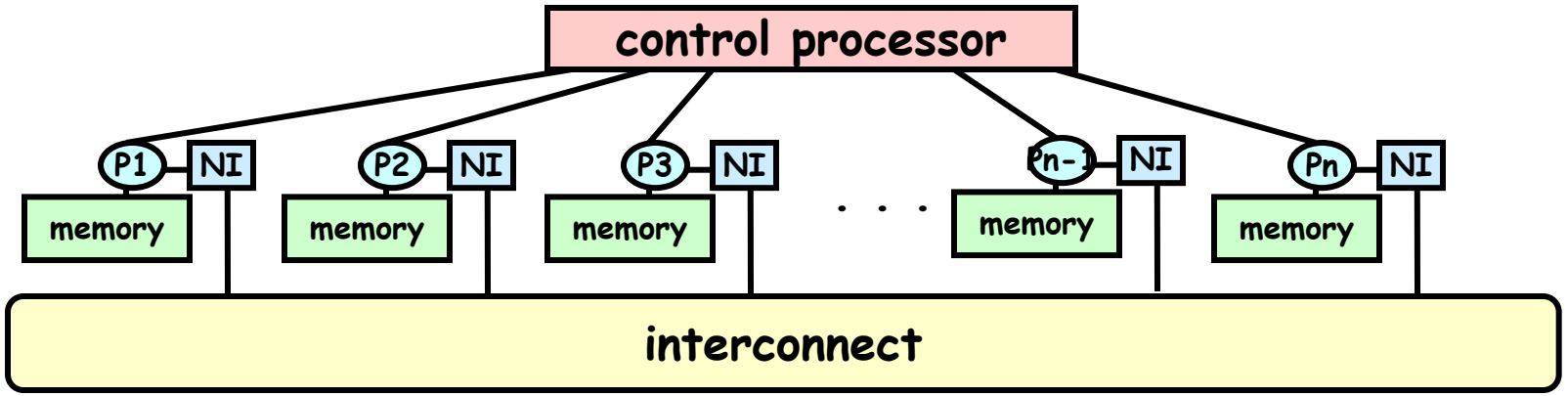
- Single thread of control consisting of parallel operations.
  - A = B+C could mean add two arrays in parallel
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously
  - Similar to Matlab language for array operations
- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines

```
A = array of all data
fA = f(A)
s = sum(fA)
```



Machine Model 3a: SIMD System

- A large number of (usually) small processors.
  - A single "control processor" issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.
- Originally machines were specialized to scientific computing, few made (CM2, Maspar)
- Programming model can be implemented in the compiler
  - mapping n-fold parallelism to p processors, n >> p, but it's hard (e.g., HPF)

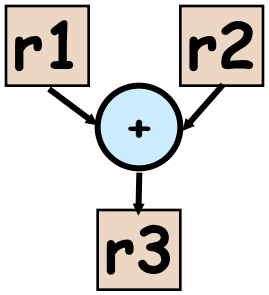


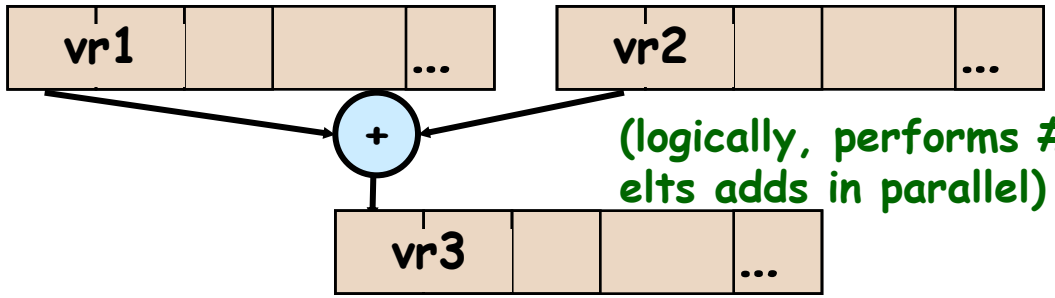
Machine Model 3b: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specific large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel
- Historically important
  - Overtaken by MPPs in the 90s
- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6), Cray X1, K
  - At a small scale in SIMD media extensions to microprocessors
    - SSE, SSE2 (Intel: Pentium/IA64)
    - Altivec (IBM/Motorola/Apple: PowerPC)
    - VIS (Sun: Sparc)
  - At a larger scale in GPUs
- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn't have to

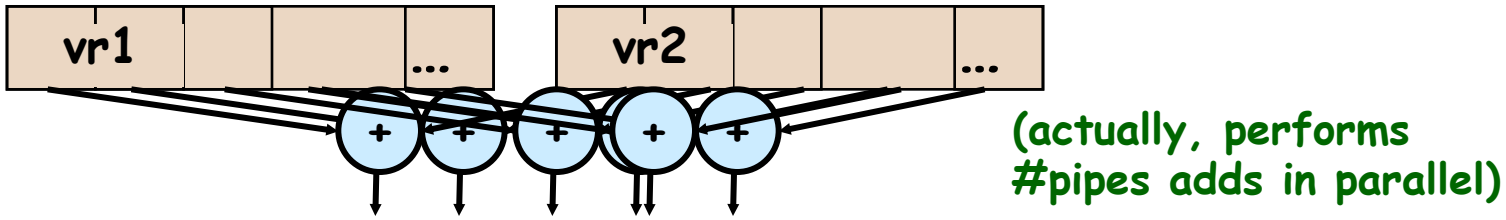
Vector Processors

- Vector instructions operate on a vector of elements
  - These are specified as operations on vector registers





- A supercomputer vector register holds ~32-64 elts
  - The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4
- The hardware performs a full vector operation in
  - #elements-per-vector-register / #pipes



Overview

Programming Models	Machine Models
1. Shared Memory	1a. Shared Memory 1b. Multithreaded Procs. 1c. Distributed Shared Mem.
2. Message Passing	2a. Distributed Memory 2b. Internet & Grid Computing
2a. Global Address Space	2c. Global Address Space
3. Data Parallel	3a. SIMD 3b. Vector
4. Hybrid	4. Hybrid

Machine Model 4: Hybrid Machines

- **Multicore/SMPs are a building block for a larger machine with a network**
- **Old name:**
  - CLUMP = Cluster of SMPs
- **Many modern machines look like this:**
  - Edison and Hopper (2x12 way nodes), most of Top500
- **What is an appropriate programming model #4 ???**
  - Treat machine as "flat", always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.
- **GPUs may also be building block**
  - Nov 2014 Top500: 14% have accelerators, but 35% of performance

Programming Model 4: Hybrids

- **Programming models can be mixed**
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice verse
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise
- **For better or worse**
  - Supercomputers often programmed this way for peak performance

What about GPU and Cloud?

- **GPU's big performance opportunity is data parallelism**
  - Most programs have a mixture of highly parallel operations, and some not so parallel
  - GPUs provide a threaded programming model (CUDA) for data parallelism to accommodate both
  - Current research attempting to generalize programming model to other architectures, for portability (OpenCL)
- **Cloud computing lets large numbers of people easily share $O(10^5)$ machines**
  - MapReduce was first programming model: data parallel on distributed memory
  - More flexible models (Hadoop, Spark, …) invented since then

Three Parallel Programming Models

- Shared address space
  - Communicate is unstructured, implicit in loads and stores
  - Natural way of programming, but can shoot yourself in the foot easily: Program might be correct, but not perform well
- Message passing
  - Structure all communication as messages
  - Often harder to get first correct program than shared address space
  - Structure often helpful in getting to first correct, scalable program
- Data parallel
  - Structure computation as a big "map" over a collection
  - Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map
  - Modern embodiment encourage, but don't enforce, this structure

Three Parallel Programming Models

- Shared address space: very little structure
  - All threads can read and write to all shared variables
  - Pitfall: due to implementation, not all reads and writes have the same cost (and that cost is not apparent in program text)
- Message passing: highly structured computation
  - All communication occurs in the form of messages (can read program and see where the communication is)
- Data parallel: very rigid computation structure
  - Programs perform the same function on different data elements in a collection

Morden practice: Mixed programming models

- Use shared address space pogromming within a multi-core node of cluster, use message passing between nodes
  - Very, very common in practice
  - User convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere
- Data-parallel programming models support shared-memory style synchronization primitives in kernels
- CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate

Summary

- Programming models provide a way to think about the organization of parallel programs. They provide abstractions that admit many possible implementations
- Restrictions imposed by these abstractions are designed to reflect realities of parallelization and communication costs
  - Shared address space machine: hardware supports any processors accessing any address
  - Message passing machine: may have hardware to accelerate message send/receive/buffering
  - It's desirable to keep "abstraction distance" low so programs have predictable performance, but want it high enough for code flexibility/portability
- In practice, you'll need to be able to think in variety of ways
  - Modern machines provide different types of communication at different scales
  - Different models fit the machine best at the various scales
  - Optimization may require you to think about implementations, not just abstractions

Thanks!