
CS 267: Introduction to Parallel Machines and Programming Models Lecture 3

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr15/

CS267 Lecture 3

1

Outline

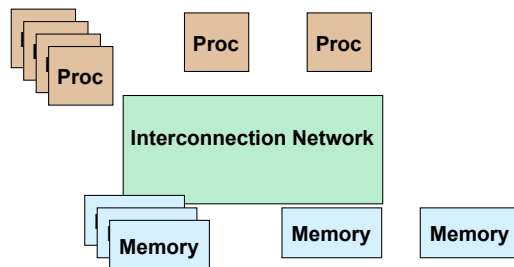
- Overview of parallel machines (~hardware) and programming models (~software)
 - Shared memory
 - Shared address space
 - Message passing
 - Data parallel
 - Clusters of SMPs or GPUs
 - Grid
- Note: Parallel machine may or may not be tightly coupled to programming model
 - Historically, tight coupling
 - Today, portability is important

01/27/2015

CS267 Lecture 3

2

A generic parallel architecture



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

01/27/2015

CS267 Lecture 3

3

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?

01/27/2015

CS267 Lecture 3

4

Simple Example

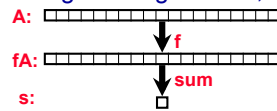
- Consider applying a function f to the elements of an array A and then computing its sum:

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

- 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
 $fA = f(A)$
 $s = \text{sum}(fA)$



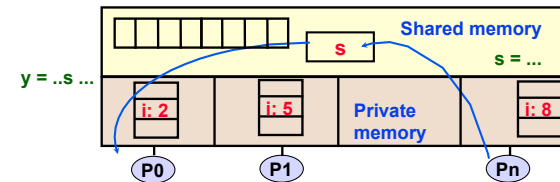
01/27/2015

CS267 Lecture 3

5

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



01/27/2015

CS267 Lecture 3

6

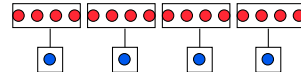
Simple Example

- Shared memory strategy:
 - small number $p \ll n = \text{size}(A)$ 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?

01/27/2015

CS267 Lecture 3

7

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

01/27/2015

CS267 Lecture 3

8

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

01/27/2015

CS267 Lecture 3

9

Improved Code for Computing a Sum

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

Why not do lock
Inside loop?

Thread 1	Thread 2
local_s1 = 0	local_s2 = 0
for i = 0, n/2-1	for i = n/2, n-1
local_s1 = local_s1 + f(A[i])	local_s2 = local_s2 + f(A[i])
lock(lk);	lock(lk);
s = s + local_s1	s = s + local_s2
unlock(lk);	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)

01/27/2015

CS267 Lecture 3

10

Review so far and plan for Lecture 3

Programming Models

1. Shared Memory

2. Message Passing

2a. Global Address Space

3. Data Parallel

4. Hybrid

Machine Models

1a. Shared Memory

1b. Multithreaded Procs.

1c. Distributed Shared Mem.

2a. Distributed Memory

2b. Internet & Grid Computing

2c. Global Address Space

3a. SIMD

3b. Vector

4. Hybrid

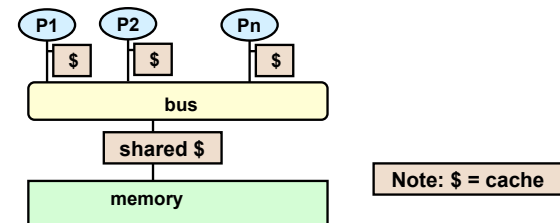
01/27/2015

CS267 Lecture 3

11

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



01/27/2015

CS267 Lecture 3

12

Problems Scaling Shared Memory Hardware

- Why not put more processors on (with larger memory?)
 - The memory bus becomes a bottleneck
 - Caches need to be kept *coherent*
- Example from a Parallel Spectral Transform Shallow Water Model (PSTSWM) demonstrates the problem
 - Experimental results (and slide) from Pat Worley at ORNL
 - This is an important kernel in atmospheric models
 - 99% of the floating point operations are multiplies or adds, which generally run well on all processors
 - But it does sweeps through memory with little reuse of operands, so uses bus and shared memory frequently
 - These experiments show performance per processor, with one “copy” of the code running independently on varying numbers of procs
 - The best case for shared memory: no sharing
 - But the data doesn’t all fit in the registers/cache

01/27/2015

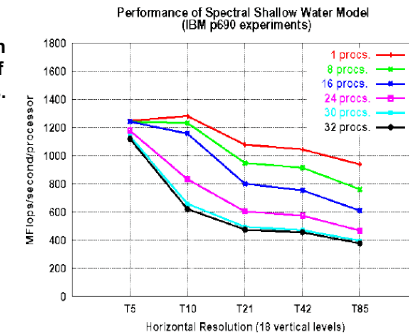
CS267 Lecture 3

13

Example: Problem in Scaling Shared Memory

PSTSWM Sensitivity to Contention

- Performance degradation is a “smooth” function of the number of processes.
- No shared data between them, so there should be perfect parallelism.
- (Code was run for a 18 vertical levels with a range of horizontal sizes.)



Process scaling on IBM p690

OAK RIDGE NATIONAL LABORATORY
U. S. DEPARTMENT OF ENERGY

UT-BATTELLE

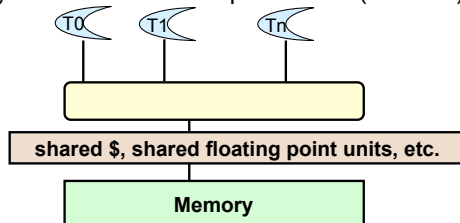
01/27/2015

CS267 Lecture 3

From Pat Worley, ORNL 14

Machine Model 1b: Multithreaded Processor

- Multiple thread “contexts” without full processors
- Memory and some other state is shared
- Sun Niagara 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)

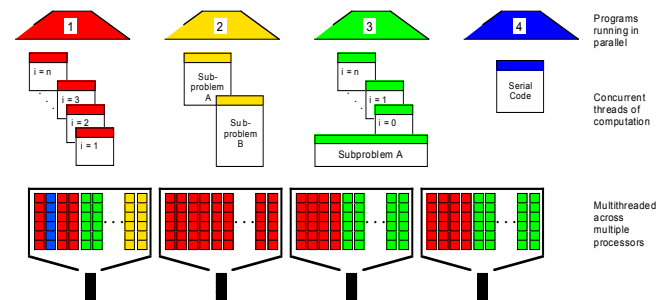


01/27/2015

CS267 Lecture 3

15

Eldorado Processor (logical view)



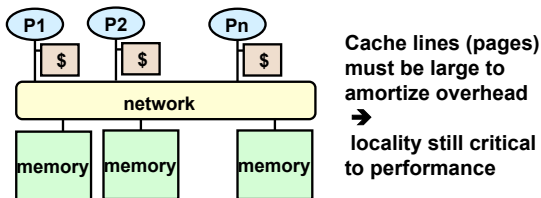
01/27/2015

CS267 Lecture 3

Source: John Feo, Cray 16

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



01/27/2015

CS267 Lecture 3

17

Review so far and plan for Lecture 3

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

01/27/2015

CS267 Lecture 3

18

Review so far and plan for Lecture 3

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

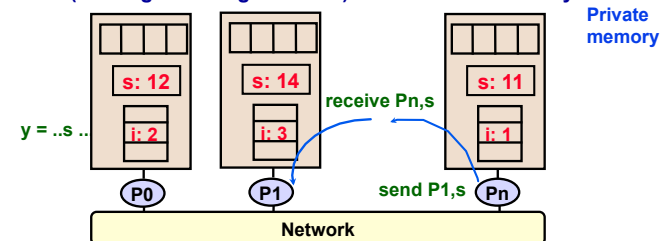
01/27/2015

CS267 Lecture 3

19

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



01/27/2015

CS267 Lecture 3

20

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

- First possible solution – what could go wrong?

Processor 1
 $x_{local} = f(A[1])$
 send x_{local} , proc2
 receive x_{remote} , proc2
 $s = x_{local} + x_{remote}$

Processor 2
 $x_{local} = f(A[2])$
 send x_{local} , proc1
 receive x_{remote} , proc1
 $s = x_{local} + x_{remote}$

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

Processor 1
 $x_{local} = f(A[1])$
 send x_{local} , proc2
 receive x_{remote} , proc2
 $s = x_{local} + x_{remote}$

Processor 2
 $x_{local} = f(A[2])$
 receive x_{remote} , proc1
 send x_{local} , proc1
 $s = x_{local} + x_{remote}$

- What if there are more than 2 processors?

01/27/2015

CS267 Lecture 3

21

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

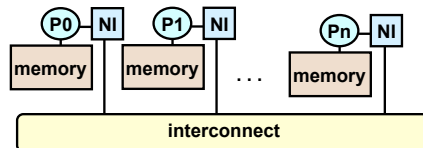
01/27/2015

CS267 Lecture 3

22

Machine Model 2a: Distributed Memory

- Cray XE6 (Hopper), Cray XC30 (Edison)
- PC Clusters (Berkeley NOW, Beowulf)
- Edison, Hopper, 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.



01/27/2015

CS267 Lecture 3

23

PC Clusters: Contributions of Beowulf

- An experiment in parallel computing systems (1994)
- Established vision of low cost, high end computing
 - Cost effective because it uses off-the-shelf parts
- Demonstrated effectiveness of PC clusters for some (not all) classes of applications
- Provided networking software
- Conveyed findings to broad community (great PR)
- Tutorials and book
- Design standard to rally community!
- Standards beget: books, trained people, software ... virtuous cycle



Adapted from Gordon Bell, presentation at Salishan
 01/27/2015 CS267 Lecture 3

24

Tflop/s and Pflop/s Clusters (2009 data)

The following are examples of clusters configured out of separate networks and processor components

- About 82% of Top 500 are clusters (Nov 2009, up from 72% in 2005),
 - 4 of top 10
- IBM Cell cluster at Los Alamos (Roadrunner) is #2
 - 12,960 Cell chips + 6,948 dual-core AMD Opterons;
 - 129600 cores altogether
 - 1.45 PFlops peak, 1.1PFlops Linpack, 2.5MWatts
 - Infiniband connection network
- For more details use “database/sublist generator” at www.top500.org

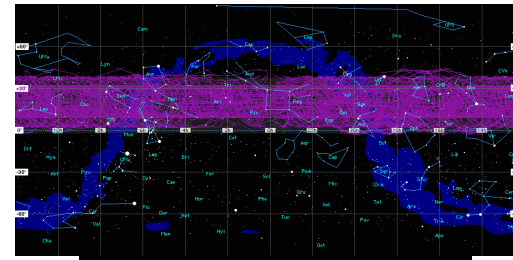
01/27/2015

CS267 Lecture 3

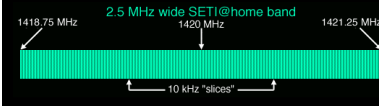
25

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



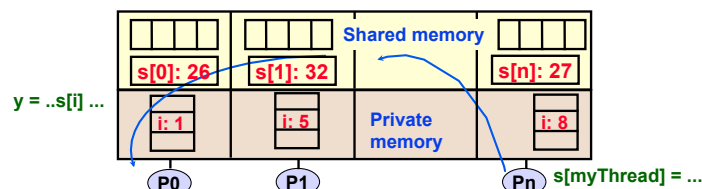
Google
“volunteer computing”
or “BOINC”

01/27/2015

26

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



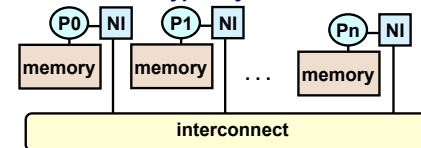
01/27/2015

CS267 Lecture 3

27

Machine Model 2c: Global Address Space

- Cray T3D, T3E, X1, and HP Alphaserver cluster
- 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

01/27/2015

CS267 Lecture 3

28

Review so far and plan for Lecture 3

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

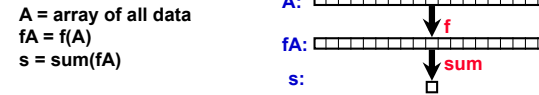
01/27/2015

CS267 Lecture 3

29

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**



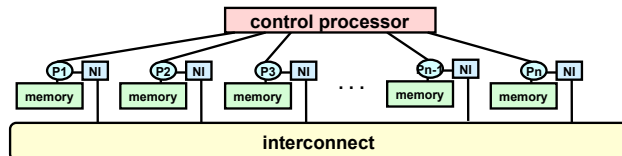
01/27/2015

CS267 Lecture 3

30

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, $n \gg p$, but it's hard (e.g., HPF)**



01/27/2015

CS267 Lecture 3

31

Machine Model 3b: Vector Machines

- Vector architectures are based on a single processor
 - **Multiple functional units**
 - **All performing the same operation**
 - **Instructions may specify 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) and Cray X1**
 - **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

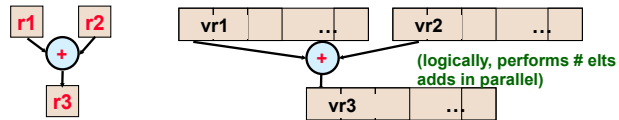
01/27/2015

CS267 Lecture 3

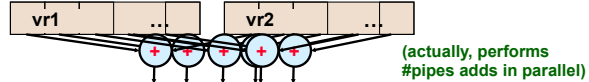
32

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



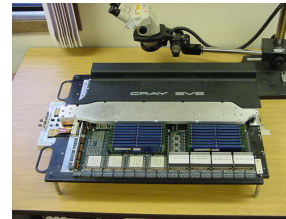
01/27/2015

CS267 Lecture 3

33

Cray X1: Parallel Vector Architecture

- Cray combines several technologies in the X1
 - 12.8 Gflop/s Vector processors (MSP)
 - Shared caches (unusual on earlier vector machines)
 - 4 processor nodes sharing up to 64 GB of memory
 - Single System Image to 4096 Processors
 - Remote put/get between nodes (faster than MPI)

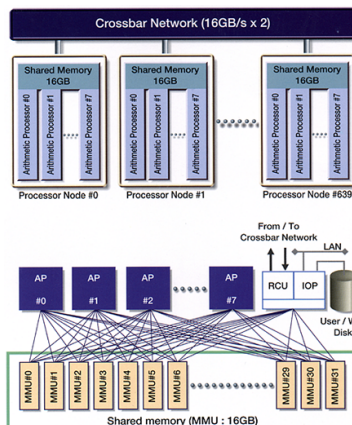


01/27/2015

CS267 Lecture 3

34

Earth Simulator Architecture



Parallel Vector Architecture

- High speed (vector) processors
- High memory bandwidth (vector architecture)
- Fast network (new crossbar switch)

Rearranging commodity parts can't match this performance

01/27/2015

CS267 Lecture 3

35

Review so far and plan for Lecture 3

Programming Models

1. Shared Memory
2. Message Passing
- 2a. Global Address Space
3. Data Parallel
4. Hybrid

Machine Models

- 1a. Shared Memory
- 1b. Multithreaded Procs.
- 1c. Distributed Shared Mem.
- 2a. Distributed Memory
- 2b. Internet & Grid Computing
- 2c. Global Address Space
- 3a. SIMD & GPU
- 3b. Vector
4. Hybrid

01/27/2015

CS267 Lecture 3

36

Machine Model 4: Hybrid machines

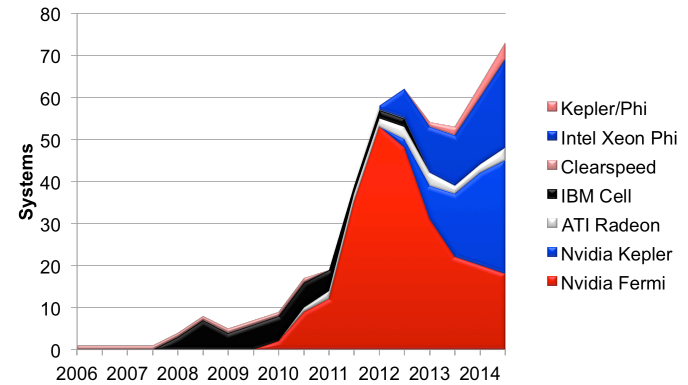
- Multicore/SMPs are a building block for a larger machine with a network
- Old name:
 - CLUMP = Cluster of SMPs
- Many modern machines look like this:
 - Edison and Hopper (2x12 way nodes), most of Top500
- What is an appropriate programming model #4 ???
 - Treat machine as “flat”, always use message 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

01/27/2015

CS267 Lecture 3

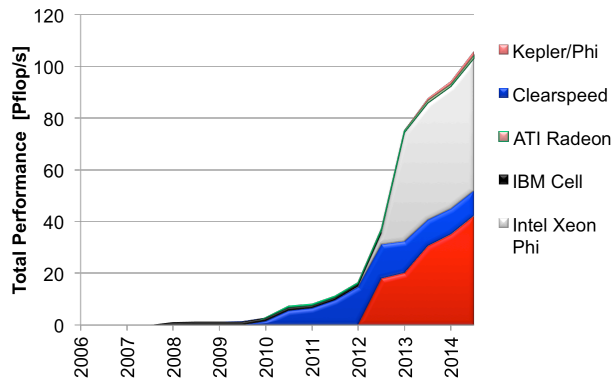
37

Accelerators in Top 500 (Nov 2014)



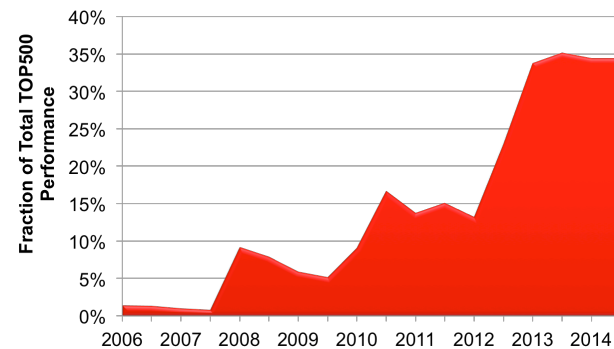
01/27/2015

Performance of Accelerators in Top500, Nov 2014



01/27/2015

Performance Share of Accelerators in Top500, Nov 2014



01/27/2015

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 versa
 - 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

01/27/2015

CS267 Lecture 3

41

Review so far and plan for Lecture 3

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 & GPU 3b. Vector
4. Hybrid	4. Hybrid

What about GPU? What about Cloud?

01/27/2015

CS267 Lecture 3

42

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)
 - Guest lecture later in the semester
- 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
 - Guest lecture later in the semester
- Both may be used for class projects

01/27/2015

CS267 Lecture 3

43

Lessons from Lecture 3

- Three basic conceptual models
 - Shared memory
 - Distributed memory
 - Data parallel and hybrids of these machines
- All of these machines rely on dividing up work into parts that are:
 - Mostly independent (little synchronization)
 - About same size (load balanced)
 - Have good locality (little communication)
- Next Lecture: How to identify parallelism and locality in applications

01/27/2015

CS267 Lecture 3

44