Chapter One &
Chapter Two (1/3)

Parallel Computers

- **Definition: "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."**

- **Questions about parallel computers:**
  - **How large a collection?**
  - **How powerful are processing elements?**
  - **How do they cooperate and communicate?**
  - **How are data transmitted?**
  - **What type of interconnection?**
  - **What are HW and SW primitives for programmer?**
  - **Does it translate into performance?**

- **The dream of computer architects since 1950s:**
  - **replicate processors to add performance**
  - **design a faster processor**

What level Parallelism?

- Uniprocessors must stop getting faster due to limit of speed of light
- Bit level parallelism: 1970 to ~1985
  - 4 bits, 8 bit, 16 bit, 32 bit microprocessors
  - Why does pursuit of bit parallelism stop at 32 bit?
  - Are 64 bit microprocessors really fast?
- Instruction level parallelism (ILP): ~1985 through 2005
  - Pipelining
  - Superscalar
  - VLIW
  - Out-of-Order execution
  - Limits to benefits of ILP?

Why Multiprocessors?

- Complexity of current microprocessors
  - Do we have enough ideas to sustain 2X/1.5yr?
  - Can we deliver such complexity on schedule?
  - Limit to ILP due to data dependency

➔ Process Level or Thread level parallelism

- Microprocessors as the fastest CPUs
  - Collecting several is much easier than redesigning new one
- Slow (but steady) improvement in parallel software (scientific apps, databases, OS)

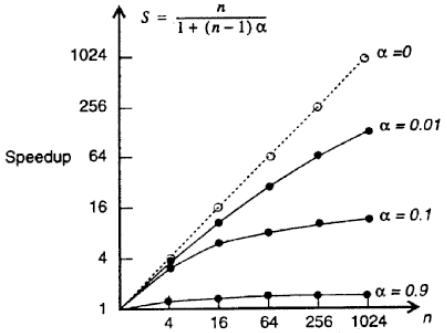
Parallel Computing

- **Amdahl's Law: The speed-up of a program is given by**

$$S_n = \frac{1}{\alpha + (1-\alpha)/n} \le \frac{1}{\alpha} \quad \text{when } n = \infty$$

- **Where, *n* = number of processors and *α* = sequential fraction of the program**

- **If *α* = 0, the maximum speed-up is *n*. However, the actual speed-up will be much less due to fixed memory size, inter-processor communication and synchronization delays.**



Flynn's Classification of Computer Systems

- **SISD (Single Instruction Single Data)**
  - **Uniprocessors**
- **MISD (Multiple Instruction Single Data)**
  - **Multiple processors on a single data stream: ???**
- **SIMD (Single Instruction Multiple Data)**
  - **Simple programming model**
  - **Low overhead**
  - **Flexibility**
  - **All custom integrated circuits: (media instructions by Intel)**
- **MIMD (Multiple Instruction Multiple Data)**
  - **Flexible**
  - **Use off-the-shelf micros**

→ Current winner: Concentrate on major design emphasis

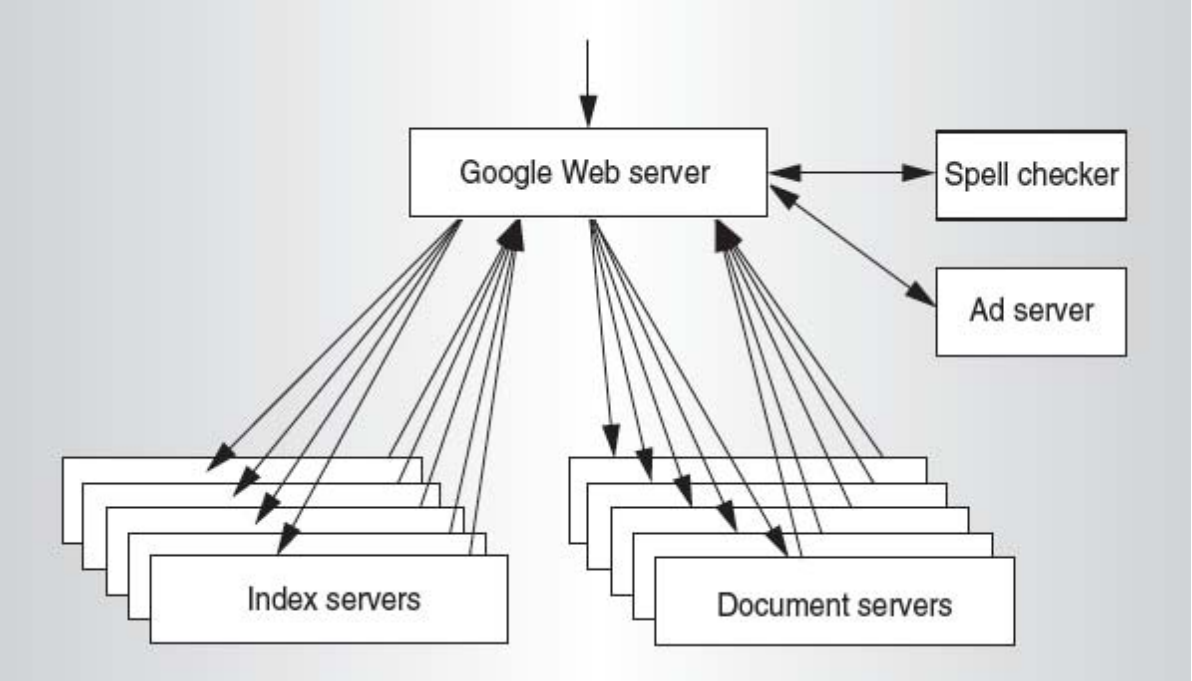
Communication Models

- **Shared Memory**
  - **Processors communicate with shared address space**
  - **Easy on small-scale machines**
  - **Advantages:**
    - Model of choice for uniprocessors, small-scale MPs
    - Ease of programming
    - Lower latency
    - Easier to use hardware controlled caching
- **Message passing**
  - **Processors have private memories, communicate via messages**
  - **Advantages:**
    - Less hardware, easier to design
    - Focuses attention on costly non-local operations
- **Can support either SW model on either HW base**

Request-Level Parallelism (RLP)

- **Hundreds or thousands of requests per second**
  - **Not your laptop or cell-phone, but popular Internet services like Google search**
  - **Such requests are largely independent**
    - Mostly involve read-only databases
    - Little read-write (aka “producer-consumer”) sharing
    - Rarely involve read–write data sharing or synchronization across requests
- **Computation easily partitioned within a request and across different requests**

Google Query-Serving Architecture



Anatomy of a Web Search

- **Google "John Hennessy"**
  - **Direct request to "closest" Google Warehouse Scale Computer**
  - **Front-end load balancer directs request to one of many arrays (cluster of servers) within WSC**
  - **Within array, select one of many Google Web Servers (GWS) to handle the request and compose the response pages**
  - **GWS communicates with Index Servers to find documents that contain the search words, "John", "Hennessy", uses location of search as well**
  - **Return document list with associated relevance score**

Problem Trying To Solve

- **How process large amounts of raw data (crawled documents, request logs, …) every day to compute derived data (inverted indicies, page popularity, …) when computation conceptually simple but input data large and distributed across 100s to 1000s of servers so that finish in reasonable time?**
- **Challenge: Parallelize computation, distribute data, tolerate faults without obscuring simple computation with complex code to deal with issues**
- **Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, Jan 2008.**

Example Applications

Application	Big Data	Algorithms	Compute Style
Scientific study (e.g. earthquake study)	Ground model	Earthquake simulation, thermal conduction, …	HPC
Internet library search	Historic web snapshots	*Data mining*	MapReduce
Virtual world analysis	Virtual world database	*Data mining*	MapReduce or HPC
Language translation	Text corpuses, audio archives,…	Speech recognition, machine translation, text-to-speech, …	MapReduce & HPC
Video search	Video data	Object/gesture identification, face recognition, …	MapReduce

There has been more video uploaded to YouTube in the last 2 months than if ABC, NBC, and CBS had been airing content 24/7/365 continuously since 1948. - Gartner

Big Data

- **Interesting applications are data hungry**
- **The data grows over time**
- **The data is immobile**
  - 100 TB @ 1Gbps ~= 10 days
- **Compute comes to the data**
- **Big Data clusters are the new libraries**

The value of a cluster is its data

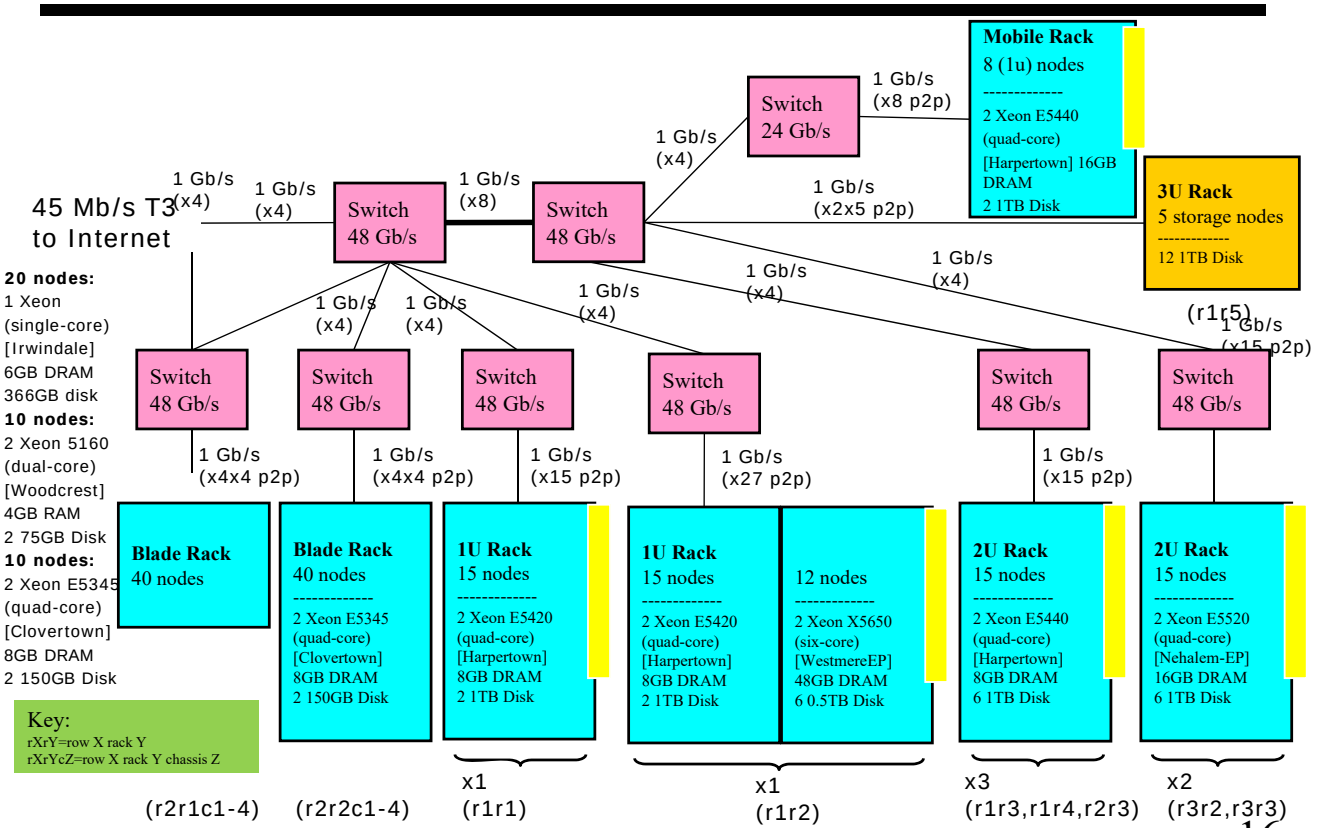
Online Processing of Archival Video

- **Research project: Develop a context recognition system that is 90% accurate over 90% of your day**
  - **Leverage a combination of low- and high-rate sensing for perception**
  - **Federate many sensors for improved perception**
  - **Big Data: Terabytes of archived video from many egocentric cameras**
- **Example query 1: "Where did I leave my briefcase?"**
  - **Sequential search through all video streams [Parallel Camera]**
- **Example query 2: "Now that I've found my briefcase, track it"**
  - **Cross-cutting search among related video streams [Parallel Time]**

Big Data System Requirements

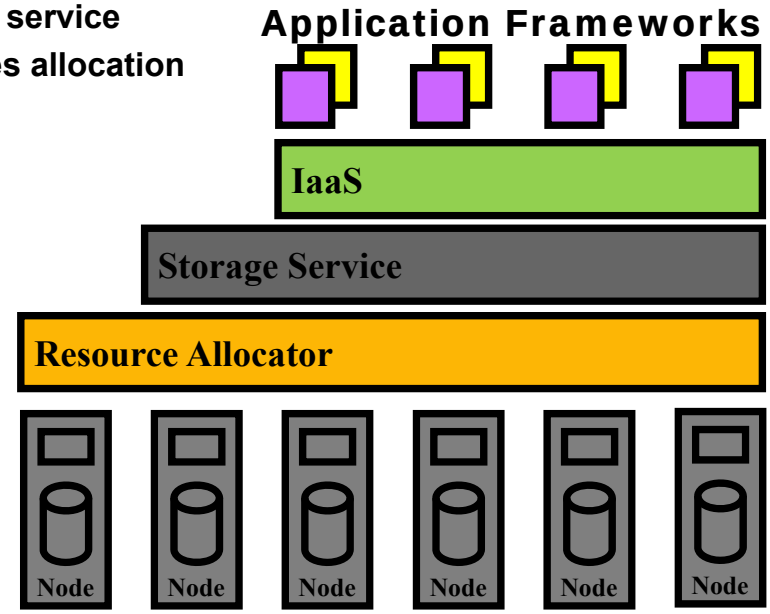
- Provide high-performance execution over Big Data repositories
  - ➔ Many spindles, many CPUs
  - ➔ Parallel processing
- Enable multiple services to access a repository concurrently
- Enable low-latency scaling of services
- Enable each service to leverage its own software stack
  - ➔ IaaS, file-system protections where needed
- Enable slow resource scaling for growth
- Enable rapid resource scaling for power/demand
  - ➔ Scaling-aware storage

Intel BigData Cluster



Cloud Software Stack – Key Learnings

- Enable use of application frameworks (Hadoop, Maui-Torque)
- Enable general IaaS use
- Provide Big Data storage service
- Enable physical resources allocation



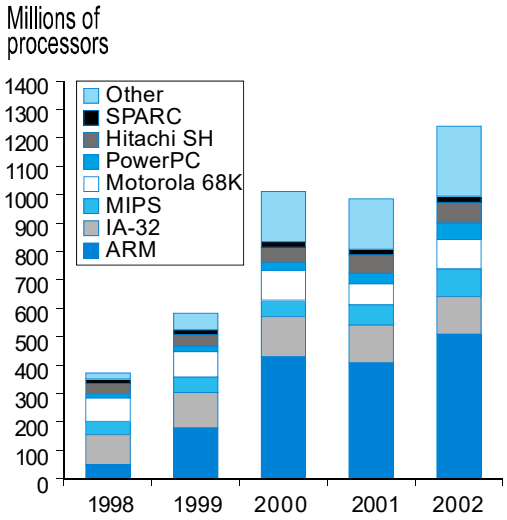
Why Physical?
1. Virtualization overhead
2. Access to phys resource
3. Security issues

MIPS Instructions

Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony



Design goals: maximize performance and minimize cost, reduce design time

MIPS arithmetic

- All instructions have 3 operands
  - Two sources and one destination
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2 ; $s0 = $s1 + $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: Simplicity favors regularity.
- Of course this complicates some things...

C code:
```
A = B + C + D;
E = F - A;
```

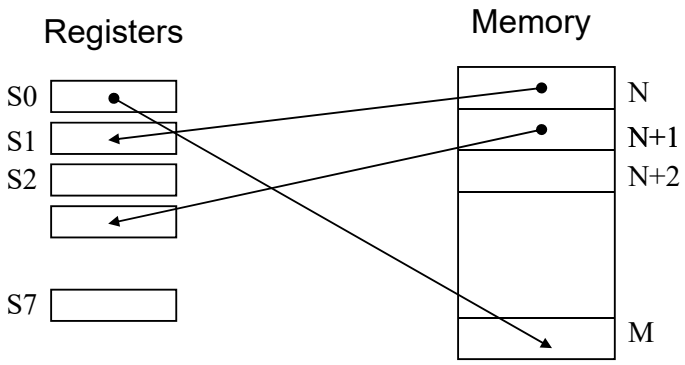
MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

$s1 = B, $s2 = C, $s3 = D, $s4 = E, $s5 = F

- Operands must be registers, only 32 registers provided
- Design Principle: Smaller is faster.

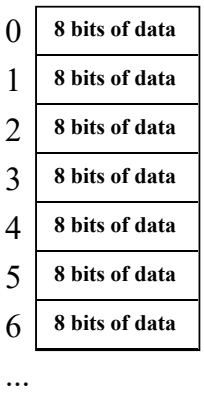
Registers vs. Memory

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables
  - the compiler tries to keep the most frequently used variables in registers and places the rest in memory
  - uses loads and stores to move variables between registers and memory.



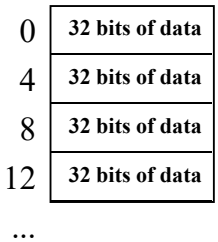
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

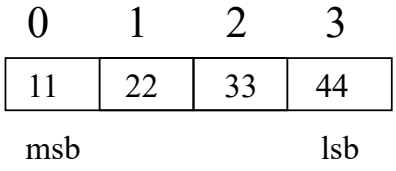


Registers hold 32 bits of data

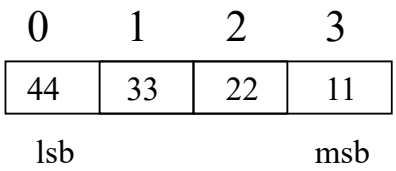
- 2^{32} bytes with byte addresses from 0 to $2^{32} - 1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32} - 4$
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Byte Order

- **Processors can number bytes within a word so the byte with the lowest number is either the leftmost or rightmost one.**
- **Big Endian (0x11223344)**



- **Little Endian (0x11223344)**



Instructions

- **Load and store instructions**
- **Example:**

C code:

```
A[8] = h + A[8];
```

MIPS code:

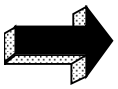
```
lw  $t0, 32($s3)    ; $t0=Mem[$s3+32]
add $t0, $s2, $t0   ; $t0=$s2+$t0
sw  $t0, 32($s3)    ; Mem[$s3+32]=$t0
```

- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

Our First Example

- **Can we figure out the code?**

```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

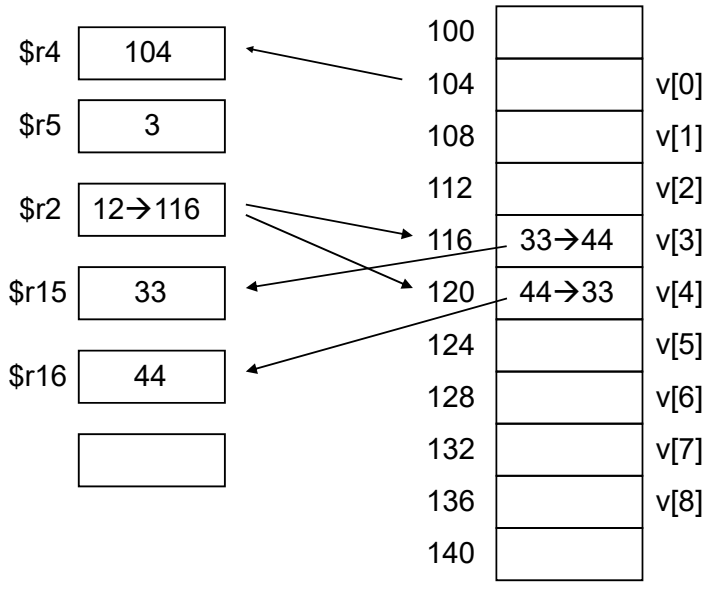


```
$r4 = v[]
$r5 = k

0($r2) = v[k]
4($r2) = v[k+1]

swap:
   muli $r2, $r5, 4
   add $r2, $r4, $r2
   lw $r15, 0($r2)
   lw $r16, 4($r2)
   sw $r16, 0($r2)
   sw $r15, 4($r2)
   jr $r31
```


Operations of the First Example

- **swap(v,3);**

```
swap:
      muli $r2, $r5, 4
      add $r2, $r4, $r2
      lw $r15, 0($r2)
      lw $r16, 4($r2)
      sw $r16, 0($r2)
      sw $r15, 4($r2)
      jr $r31
```



So far we've learned:

- **MIPS**
  - **loading words but addressing bytes**
  - **arithmetic on registers only**

- **Instruction** **Meaning**

```
add $s1, $s2, $s3       $s1 = $s2 + $s3
sub $s1, $s2, $s3       $s1 = $s2 – $s3
lw $s1, 100($s2)        $s1 = Memory[$s2+100]
sw $s1, 100($s2)        Memory[$s2+100] = $s1
```