WSC

Strong scaling - adding more machines make algorithm faster on a given data
Weak Scaling - adding more machines lets you process more data in the same amount of time
Data Level Parallelism - independent data sets with independent processing.
Big Idea - instead of using expensive super computers use 10,000 - 100,000 cheeper servers & networks
· emphasize cost efficiency
· attention to power: distribution & cooling
· (relatively) homogenous hardware/software, which is good for error handling and scaling
Quick Facts:
· CAPEX - cost to buy equipment (buy servers)
· OPEX - cost to run equipment (electricity used)
· servers cost most $$$ with replacement every 3 years
· building cooling increase PUE
· a lot of servers sit idle (most servers are at 10% - 50% max load) and waste energy
· ultimate goal is energy proportionality - %peak load = %peak energy
Power Usage Effectiveness (PUE) $= \frac{TotalBuildingPower}{ITPower}$
can never be less than 1
tells us nothing about the efficiency of our computing hardware, only the relative efficiency of our IT equip vs. non IT equip.

MapReduce

Map - slice data into shards, distribute to workers, compute sub-problem
map(inKey, inVal) → list(outKey, intermedVal)
Combiner - compress the output of a single mapper to save on bandwidth and to distribute work more evenly
apply reduce operation distributed across map tasks, producing set of intermediate pairs
Reducer - collect and combine subproblems, combining all intermediate values for a particular key to produce a set of merged output values
all mappers must finish before any reducer can start
Data Shuffle/Sort (taken care of by framework) merges all intermediate values with same key from mapper output to produce key value pairs of
(intermedKey, list[all associated values])
reduce(outKey, list(intermedVal)) → list(outVal)

C Programming Language

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	int	2	-32768 to 32767
Short Integer	short int	2	-32768 to 32767
Long Integer	long int	4	-2,147,483,648 to 2,147,438,647
Unsigned Integer	unsigned int	2	0 to 65535
Unsigned Short integer	unsigned short int	2	0 to 65535
Unsigned Long Integer	unsigned long int	4	0 to 4,294,967,295
Float	float	4	1.2E-38 to
Double	double	8	2.2E-308 to
Long Double	long double	10	3.4E-4932 to 1.1E+4932

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

C arrays are almost identical to pointers
array variable is a pointer to the first (0^{th}) element
arr[0] = *arr
arr[2] = *(arr+2)
must pass array & size because array in C does not know its own length
arr[i] = *(arr + i)

MIPS

registers are 32 bits wides (this quantity is called a *word*), 4 bytes
memory addresses are really in bytes, not words   word addresses are 4 bytes apart
jal - jumps to address and simultaneously saves address of following instruction in register $ra
bible of MIPS - the green MIPS sheet!
data transfer object: constant(register used to access memory) → sum the memory address
address in a word match the address of 1 of the 4 bytes within that word
addresses of sequential words differ by 4 bytes
MIPS words must start at addresses that are multiples of 4
e.g. *beq $t0, $0, 2* ← means 2 instructions, each instruction is 4 bytes (32 bits)

C Memory Model

· **Stack**
· local variables, broken into stack frames, grows down
· **Heap**
· stores data for which space was allocated with malloc(), grows upwards
· **Static**
· global, static variables, & string literals
· fixed size during execution
· **Code**
· contains instructions, fixed size during exec

High Level → Compiler → Assembly Language → Assembler → Binary Instructions

Below the Program

Operating System

interfaces between user's application & hardware
· handles input/output operations
· allocates memory & storage
· provides sharing of computer among multiple applications

Compiler

a program that translates high level language statements into assembly language statements, symbolic representations of machine instructions

Assembler

input - assembly language code
output - object code
reads & uses directives, replaces pseudo instructions, produces machine language, & creates object files
assembler directions:
· .text - subsequent items put in user text segment
· .data - subsequent items put in user data segment
· .globl sum - declares sum global and can be referenced from other files
· .ascii str - store the string str in memory and null terminate it
· .word w_1 - store a s32 bit quantity in successive memory words

Object Module

assembler or compiler translates program into machine instructions
provides information for building a complete program from the pieces:
· header: contents of object module
· text: translated instructions
· static data segment: data allocated for life of program
· relocation info: for contents that depend on absolute location of program
· symbol table: global definitions and external references
· debug info: for associating with source code

Linker

produces executable image
· merges segments
· resolves labels
· patches location dependent and external references

Loader

· reads header to determine segment sizes
· creates virtual address space
· copy text and initialized data into memory
· sets up arguments on stack
· initializes registers   · jumpstarts routine
(interpreters are sower than compiled programs, but useful for debugging)

Underlying Hardware

Five Components

· input: mechanism by which comp is fed info
· output: mechanism by which computer conveys result of computation
· memory - storage area in which programs are kept when running and area containing data necessary by program to run
· data path - component of the processor that performs arithmetic operations
· control - component of the processor that commands the data path, memory, & I/O devices according to the program instructions

Floating Point

single precision: 1 sign bit (31), 8 exponent bits (30-23), 23 fraction bits (22-0)
FP = $(-1)^S \times (1 + F) \times 2^{E-bias(127)}$
special values:
· $\pm$ Zero: E = 0, F= 0
· NaN: E = 255 (all 1's), F $\neq$ 0
· $\pm\infty$: E = 255 (all 1's), F = 0
· Denormalized: E = 0, F $\neq$ 0
underflow: negative exponent is too large to fit in exponent field
overflow: exponent is too large to be represented in exponent field

Performance

throughput/bandwidth = total amount of work done in a given time
to maximize performance, minimize response/execution time
Performance = $\frac{1}{executiontime}$
$\frac{performance_x}{performance_y} = \frac{executiontime_y}{executiontime_x} = n, \rightarrow$ x is n times faster than y
response time - time to complete a task
CPU execution time = time CPU spends computing a task not including time spent waiting for I/O
User CPU time - CPU time spent in the program itself
System CPU time - CPU time spent in the operating system performing tasks on behalf of the program
clock cycles - discrete time intervals determining when events take place in the hardware
clock period - time for a complete clock cycle
clock rate - inverse of the clock period

Performance Equations

CPU execution time = CPU clock cycles for a program $\times$ Clock Cycle time = $\frac{CPUclockcyclesforaprogram}{clockrate(\frac{1}{clockcycletime})}$
CPU clock cycles = instructions for a program $\times$ average clock cycle per instruction
CPI (clock cycles per instruction) - average number of clock cycles per instruction for a program, an average of all instructions executed in a program.
CPU time = instruction count $\times$ CPU $\times$ clock cycle time = $\frac{instructioncount \times CPI}{clockrate}$
CPI = $\frac{CPUclockcycles}{instructioncount}$
Time = $\frac{seconds}{program} = \frac{instructions}{program} \times \frac{clockcycles}{instruction} \times \frac{seconds}{clockcycle}$

Cache

AMAT (avg. mem. access time) = hit time + miss rate $\times$ miss time
local miss rate = fraction of references to one level of cache that miss, e.g. local miss rate of L2 = $\frac{L2misses}{L1misses}$
global miss rate = fraction of references that miss in all levels of a multi-level cache
Tag: tells us where it came from in memory.
Tag bits = address size - index - offset
Index: which row/block to look at in cache.
Index bits = $\log_2(\frac{cachesize}{blocksize})$, in other words log of the number of blocks
Offset: which column of cache to look at.
Offset bits = $\log_2$(block size (bytes))
bits per row = valid bit + dirty bit (write back cache only) + data (block size) + tag
cache size = $2^{offsetbits+indexbits}$
number of blocks = cache size $\div$ block size
tag bits = $\log_2(\frac{memorysize}{cachesize})$
$\text{CPI}_{stall} = \text{CPI}_{ideal}$ + data miss cycles + instruction miss cycles $\text{CPI}_{stall} = \text{CPI}_{ideal}$ + L1 instruction miss cycles + L1 data miss cycles

Types of Instructions on Types of Data

SISD: Single instruction, single data stream, e.g. Intel Pentium 4
MISD: Multiple instruction, single data stream, e.g. none
MIMD: Multiple instruction, multiple data stream, e.g. Intel Xeon e5345
SIMD: Single instruction, multiple data stream, e.g. SSD Instructions of x86
SIMD is weakest in case or switch statements, where each execution unit must perform a different operation on its data.

Amdalhs Law

Maximum speedup from parallelism = $\frac{1}{((1-P)+\frac{P}{N})}$
P = proportion of program parallelizable N = number of cores

Signed vs. Unsigned MIPS

sra - shift right arithmetic: sign extension! right shift for signed quantities
srl - shift right logical: zero extension! right shift for unsigned quantities
sll - shift left logical: zero extension! left shift for signed & unsigned quantities

Recursive MIPS Example

```
Int sum (int n)
{
    return n ? n + sum(n –1): 0;
}
```

```
sum:
addi $sp, $sp,-8 # allocate space on stack
sw $ra, 0($sp) # store the return address
sw $a0,4($sp)
# store the argument slti $t0, $a0, 1
# check if n > 0
beq $t0, $0, recurse # n > 0 case
add $v0, $0, $0 # start return value to 0
addi $sp, $sp, 8 # pop 2 items off stack
jr $ra # return to caller

recurse:
addi $a0, $a0, - 1 # calculate n-1
jal sum # recursively call sum(n - 1)

lw $ra, 0($sp) # restore saved return address
lw $a0, 4($sp) #restore saved argument
addi $sp, $sp, 8 #pop 2 items off stack
add $v0, $a0, $v0 # calculate n + sum(n - 1)
jr $ra #return to caller
```