CS61C Lecture Notes

* **Flynn Taxonomy + Data-level Parallelism (10/29)**
  + **Single-Instruction/Single-Data Stream (SISD)**
    - Single processor with no parallelism and no speedup
  + **Single-Instruction/Multiple-Data Stream (SIMD)**
    - Single processor works on independent data sets (natural parallelism)
  + **Multiple-Instruction/Multiple-Data Stream (MIMD)**
    - Multiple autonomous processors executing different instructions on different data sets
  + **Amdahl’s Law**
    - Parallelism speeds up a fraction of the task by a factor of S, remainder of the task is unaffected
    - Speedup = ( (1-F) + F/S )-1 (S=processors, F=fraction)
    - Execution time = normal execution time \* (1-F + F/S)
    - Strong scaling: speedup without increasing size of problem   
      Weak scaling: speedup due to increasing problem size to account for number of available processors
    - Load balancing: give each processor the same amount of work
  + **SIMD Instruction Architecture** 
    - Uses larger registers and “packing” to be able to load more data in a single cycle and apply a single instruction to all the data within the register
    - Compiler changes iteration rate at assembly level by unrolling loop and adding overhead per iteration
  + **Intrinsics** 
    - C functional procedures for inserting assembly language into C code, maximizing speedup
* **Amdahl’s Law, Thread-level Parallelism, OpenMP Intro (11/3)**
  + Serial sections of code limit speedup
  + Intel SSE SIMD Instructions use 128 bit registers to exploit data-level parallelism in loops
  + **Multiprocessor Execution** 
    - Allows for increased throughput (job-level parallelism)
    - Processors communicate via shared memory on independent instructions
    - Improves run time of single program if designed for multiprocessor
  + **Threads**
    - A sequential flow of instructions from one “job”
    - Each thread has its own PC + register and access to shared memory/caches
    - OS responsible for feeding software threads to hardware (harts)
      * Can pull an ongoing thread and save its “state” and activate a new thread while waiting for another thread’s access to memory (300 clock cycles) or network access (thousands of clock cycles) to come back
  + **OpenMP**
    - Language extension for multithreaded, shared memory parallelism
    - Shared vs. Private variables
      * Shared: all threads have access to same variable (no storeback for current result)
    - The number of threads is equal to the number of cores available to run the program on (hardware dependent)
  + **Data Races**
    - When two memory accesses from different threads to same location with at least one being a write… danger is that one or both threads get an incorrect value (look up examples)
    - Threads fork and join when they reach parallelizable code
* **Cache Coherence, OpenMP sharing issues, Performance (11/5)**
  + Review
    - SIMD + MIMD allow for higher performance through data-level parallelism and thread-level parallelism respectively
    - With multithreading, must coordinate among threads to prevent data races (locking shared data using load-linked/store-conditionals)
  + **Fork-Join Model**
    - OpenMP programs begin as a single “master thread” and execture sequentially till they reach a parallel region where they can fork (a team of parallel threads which execute instructions independently) the join (on completion of parallel region, threads terminate to leave only master thread)
    - Only works when no early exits from loop are present (break/return/exit)
  + **OpenMP Commands**
    - #pragma omp parallel: each thread runs a copy of code being paralleled (redundant)
    - #pragma omp parallel for: breaks for loop into chunks for each thread
    - #pragma omp parallel for private(var) reduction(+:sum): does the reduction operation to all private copies of sum to yield a correct master shared var value
  + **Multiprocessor Cache** 
    - Each core has a private cache holding data it has recently accessed to reduce bandwidth demands on memory caches, because an “incorrect” or not updated value is in another private local cache
      * Solution: when a processor writes to memory, notifies other processor that their copies of data are invalid through interconnection network
      * If processors with the invalid data try to read again, they look at the proper cache with updated values and load it
        + **False sharing**: a block Ping-Pong effect occurs when two processors are accessing different variables but same “block” in cache (ie. w/ 32 byte block, mem accesses to 4000 and 4012)
    - **Cache misses**: compulsory, capacity, conflict, and coherence misses
      * Coherence misses can make up majority of misses in a parallel program if block size is too big because there’s a bigger chance of invalidating blocks in other caches (False sharing)
* **Warehouse Scale Computing, MapReduce, Spark (11/12)**
  + Server, Rack, Array
    - 1 server = 8 cores, 16gb RAM, 4 Tb disk
    - 1 rack = 40-80 servers, 1-10 Gbps, Ethernet at $30/gbps/server
    - 1 array = cluster of 16-3 racks, expensive switch
    - Advantage of WCS storage hierarchy: lower latency to RAM of another server than local disk but higher bandwidth to local disk than RAM in another server
  + Power Usage
    - Power Usage Effectiveness (PUE): Total building power/IT Equipment Power
      * Ideal PUE = 1.0; Google’s PUE = 1.2; Average PUE = 1.83
    - System efficiency approaches ideal w/ high utilization
  + Request-level Parallelism
    - Most requests are to read-only databases
    - Computation is partitioned easily across different requests/same request
    - Anatomy of a web search:
      * 1. Direct request travels to nearest Google WCS ­­🡪 Array 🡪 Server 🡪 Document List w/ associated relevance scores
      * 2. Ad system, web page is created using document list and adds
    - Redundant copies of data increase parallelism and tolerance for failure
  + Data-level Parallelism
    - MapReduce and scalable file systems on WSC
      * MapReduce is a programming model that addresses a type of problem that comes up often Computation specified in terms of a map function and reduce function
      * Map slices data into key-value pairs and distributes to parallel computers to compute result
      * Apache Hadoop and Spark are two common implementations for large-scale data processing
  + Mean time to failure = meant time to repair \* availability of hosting service (fraction)
* **OS Support, Base and Bounds, Interrupts, Virtual Memory Intro (11/17)**
  + Instruction Set Architechture for I/O
    - Processor needs to read or write a sequence of bytes for Input/OutputMIPS processor uses normal loads and stores for I/O
      * A portion of address space is dedicated to I/O communication paths
      * Memory Mapped Input/Output
    - Input is relatively slow compared to processor speed (ie. keyboards)
      * MIPS uses a control and data register with devices
      * Control is set to “ready” by I/O device and processor loads/stores data
    - Polling
      * Processor continuously checks that I/O is ready
      * For some devices, can be excessively slow
      * Alternative: exception mechanism interrupts programs when I/O ready, return when data transfer is over (requires saving of user registers and current PC to jump back to once exception is handled)
      * Low data transfer/s rate
  + Base and Bounds Registers
    - Software level outputs some address to read/write from/to and the OS adds the base register to it (the base signifies the start address of the memory allocated for that program) and ensures that the resulting address is not past the value in the bounds register (marks the end of the memory allocated for that program), otherwise segmentation fault
    - Problem: storage fragmentation, smaller and smaller pieces result
  + Paged Memory Systems
    - Page table per program makes it possible to store the pages of a program’s instructions non-contiguously (unaligned pieces of memory all think they’re aligned)
    - Processor-generated address split into virtual page number and offset
    - Page Tables allow programs to refer to slices of memory allocated for them
* **Virtual Memory 2 (11/19)**
  + Terminology
    - Interrupt: event external to current program (I/O); asynchronous to program
    - Exception: event during execution of an instruction of current program
    - Trap: action of servicing interrupt/exception w/ a hardware jump to code that will handle the trap (exceptions more complex than interrupt)
    - Internal pipeline creates problems with handling exceptions (control hazards?)
      * Exception flags are ignored until a commit point (memory stage in pipeline, where we change the state of the machine)
      * Exceptions in earlier instructions (or pipe stages) override exceptions from later instructions (or for a single instruction)
      * If a trap occurs, update cause + EPC registers (holds the PC to jump back to after the trap), kill all stages, and change PC to handler code
  + Why is VM necessary?
    - Connects memory and disk
    - Simplifies memory for applications and protects between processes (prevents memory fragmentation because there are different block sizes required for each app and blocks are allocated using a base through a bounds register)
    - Allows for more total memory than what is available at the HW level for apps
  + Address spaces
    - Virtual Address space: the set of addresses that user program knows about
    - Physical address space: the set of addresses that map to actual physical cells in memory (only OS accesses this)
      * Physical memory essentially acts as a cache of pages for currently running programs
  + Blocks and Pages (and Words and Bytes) are just ways of looking at memory and dividing it up (abstraction)… Pages > Blocks > Words > Bytes (in terms of size)
  + Page tables
    - Each program has its own page table that translates between virtual memory accesses and physical addresses
    - Page tables require a space proportional to address space, number of programs, etc., and are thus too big for registers (stored in main memory)
    - Page tables contain Page Table Entries (PTEs)
      * PTEs composed of 1 bit to indicate if page exists (valid bit) and Physical Page Number (PPN) or Disk Page Number (DPN), and status bits for protection and usage (read, write, execute)
  + Accessing Pages
    - If we access a page not in RAM w/ a load or store, if it exists (on the disk) it gets transferred to RAM, otherwise OS assigns an unused page for use
      * If RAM is full, we replace the least used page currently in RAM (put it on disk) with the page
    - Linear pages take too much memory so we use 2 level page tables w/ 2 indices contained within the PTE
    - Since each access to an index to get the page table is a memory access (expensive), we use a Translational Look-Aside Buffer (TLB), which is a fully associative cache that contains recent VM-Physical translations (Key: Virtual address, Result: Physical address, then a mem access like normal w/ caching)
      * Replacement policy on conflict misses: Random/First in First Out (FIFO)
* **IO, DMA, Disks, Parity and Hamming ECC**
  + Non-volatile storage: retains its value without applying power to disk
  + Magnetic disks are used to keep a working file system w/ long term backup *and* create a secondary store for main-memory (large, inexpensive, slow level in mem hierarchy)
  + DMA is beneficial when large chunks of data need to be transferred *FROM* an IO device to the computer’s RAM without consuming a lot of the CPU (so the DMA controller does it)
    - Must explicitly manage coherency if DMA puts into RAM and not caches
    - Messes up CPU’s working set if put between CPU and L1 Cache
  + Parity bits are set to create even parity (an even number of ones) for each group and the number of parity bits (along with where they go) is dependent on the number of bits in the original word
    - **Position 1** checks 1, 3, 5, 7, 9, 11
    - **Position 2** checks 2, 3, 6, 7, 10, 11
    - **Position 4** checks 4, 5, 6, 7, 12
    - **Position 8** checks 8, 9, 10, 11, 12