CSCI-GA.3033-017 Special Topic: Multicore Programming Homework 1
Due October 2, 2017

1. Warmup: Discuss five types of parallelism (hint: the lowest-level type is instruction-level parallelism). What are they, and for each one, what is necessary from the hardware and/or programmer? There are more than five types, but make sure you get most or all of the types that involve a single computer.

Instruction level parallelism (ILP): Processors which use pipelining to execute instructions are called ILP processors. The execution of each instruction is partitioned into several steps which are performed by dedicated hardware units (pipeline stages) one after another. The advantage is that the different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed.

– Multiple instructions from the same instruction stream can be executed concurrently
– Generated and managed by hardware (superscalar) or by compiler (VLIW)
– Limited in practice by data and control dependences

Loop level parallelism: Many algorithms perform computations by iteratively traversing a large data structure. The iterative traversal is usually expressed by a loop provided by imperative programming languages. A loop is usually executed sequentially which means that the computations of the ith iteration are started not before all computations of the (i − 1)th iteration are completed. This execution scheme is called sequential loop in the following. If there are no dependencies between the iterations of a loop, the iterations can be executed in arbitrary order, and they can also be executed in parallel by different processors. Such a loop is then called a parallel loop. Depending on their exact execution behavior, different types of parallel loops can be distinguished

Thread-level / task-level / function level parallelism (TLP): Task parallelism involves the decomposition of a task into subtasks and then allocating each subtask to a processor for execution. The processors would then execute these subtasks simultaneously and often cooperatively. Task parallelism does not usually scale with the size of a problem.

– Multiple threads or instruction sequences from the same application can be executed concurrently
– Generated by compiler/user and managed by compiler and hardware

– Limited in practice by communication/synchronization overheads and by algorithm characteristics

Data level parallelism (DLP): Data parallelism is a form of parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel.
– Instructions from a single stream operate concurrently on several data
– Limited by non-regular data manipulation patterns and by memory bandwidth

Transaction level parallelism:
Multiple threads/processes from different transactions can be executed concurrently. It is limited by concurrency overheads.

Parallelism by multiple functional units: Many processors are multiple-issue processors. Different independent instructions can be executed in parallel by different functional units. The number of functional units that can efficiently be utilized is restricted because of data dependencies between neighboring instructions.

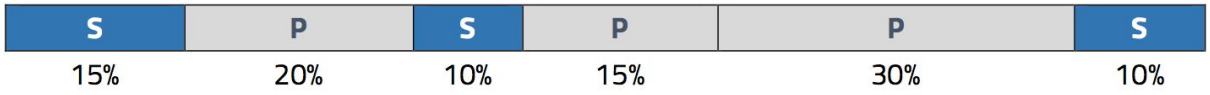
Parallelism at bit level: the word size used by the processors (now a word size of 64-bits). This development has been driven by demands for improved floating point accuracy and a larger address space.

2. Without Googling (although the slides are fair game), what makes MISD processor architectures unusual? What are they generally used for, and why aren't they used more widely?

In computing, MISD (multiple instruction, single data) is a type of parallel computing architecture where many functional units perform different operations on the same data. It is unusual because MISD can be seen as you compute the same thing multiple different ways.

Fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors, in a manner known as task replication, may be considered to belong to this type. Not many instances of this architecture exist, as MIMD and SIMD are often more appropriate for common data parallel techniques. Specifically, they allow better scaling and use of computational resources than MISD does. However, one prominent example of MISD in computing are the Space Shuttle flight control computers.

3. Apply Amdahl's law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 16, and (f) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

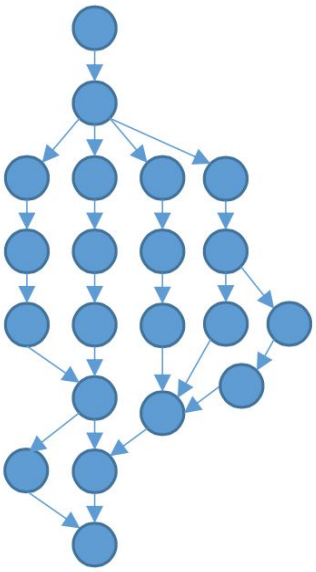


Amdahl's Law: 1/(F+(1-F)/P)
F (the fraction of a calculation that is sequential) = 0.35

a) 1 / (0.35 + 0.65/1) = 1
b) 1 / (0.35 + 0.65/2) = 1.48148148148
c) 1 / (0.35 + 0.65/4) = 1.9512195122
d) 1 / (0.35 + 0.65/8) = 2.31884057971
e) 1 / (0.35 + 0.65/16) = 2.56
f) 1 / (0.35 + 0.65/∞) = 2.85714285714

4. Assuming that each block of the diagrammed program takes 1 unit of time, what is the work and span of the following program? What is the parallelized execution time TP on (a) P=1 processors? (b) P=5 processors? (c) P=6 processors? (d) How long is the critical path?

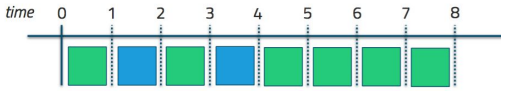
Work = 21
Span = 9
a) Tp = 21 b) Tp = 9 c) Tp = 9 d) Length of critical path: 7



5. Explain the difference between concurrency and parallelism with an example: if an operating system is executing two long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on two cores, or in parallel on two cores? Comment on running the programs in parallel on one core.

Concurrency means that two or more calculations happen within the same time frame, and there is usually some sort of dependency between them. Concurrent *(occurring or existing simultaneously)* implies that different code **MAY** execute at the exact same cycle. It means that things can **possibly** happen in parallel if multiple processors or a processor with multiple cores is available and the program is crafted correctly. Just adding threads does not imply concurrent execution. The reason we say **MAY** and **possibly** is that anytime the programs separate threads need to share volatile/mutable state, other threads that need access to that state can not continue executing and will have to wait their turn to access that state, and things start happening serially again.

Example: execute two programs(blue and green) concurrently on one core



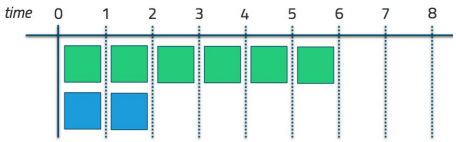
Example: execute two programs(blue and green) concurrently on two cores

Time

1	2	3	4	5	6
					

Parallelism means that two or more calculations happen simultaneously. Parallelism is one way to implement concurrency, but it's not the only one. Another popular solution is interleaved processing (a.k.a. coroutines): split both tasks up into atomic steps, and switch back and forth between the two. Running the programs in parallel on one core is not plausible.

Example: execute two programs(blue and green) in parallel on two cores



6. Broadly, what's the point of cache? Does its purpose differ between single- and multi-core processors (if so, how)? Does its implementation differ between single- and multi-core processors (if so, how and why)?

Cache memory, also called CPU memory, is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. This memory is typically integrated directly with the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU. Cache memory provides faster data storage and access by storing an instance of programs and data routinely accessed by the processor. Thus, when a processor requests data that already has an instance in the cache memory, it does not need to go to the main memory or the hard disk to fetch the data.

The problem with memory is that small is quick, large is slow. If you have a lot of memory, it takes a lot of time to access it. So we have several levels of caches between the processor and the main memory. L1 cache is generally around 16–128KiB and takes a couple of processor cycles to access. L2 cache usually has a capacity of 256KiB and takes tens of cycles to access. L3 cache is usually high single digit MiB to tens of MiB, also usually shared by all cores, and takes somewhere in the range of 20 and 100 cycles to access.

7. What are temporal and spatial locality, for example as they apply to caches? Which one(s) is/are more likely to come into play with frequent reads and writes to the elements in a 1x1-element matrix? A 1000x30-element matrix?

Temporal Locality: The concept that a resource that is referenced at one point in time will be referenced again sometime in the near future.

Spatial Locality: The concept that likelihood of referencing a resource is higher if a resource near it was just referenced.

Temporal Locality is more likely to come into play with frequent reads and writes to the elements in a 1x1-element matrix.

Spatial Locality is more likely to come into play with frequent reads and writes to the elements in a 1000x30-element matrix.