**Java**: **Thread**() object, using **Runnable** object with **run**() entry; **start**() to return in current thread and invoke **run**() in new thread, **join**()ed threads are not garbage collected (can revive). Use **static** for shared variables. **Race condition**: order of thread exec leads to different results. **Thread safe**: functions can be called concurrently from multiple threads and produce correct results. **Synchronization**: makes code thread safe (use *synchronized* around race sections – same as openmp block). **Atomic**: sequence of ops is uninterruptable. **Volatile**: all reads and writes are atomic (cannot be interrupted in reading/writing, but seq. of read/write can be). Atomics are best for single ops (must read/write to memory on every op). Synchronized blocks are more flexible/better for multiple ops (keep variables in registers during block, write at end), but serial. Typical Java design: || objects of same class, run each object in separate thread, synchronize on class (each thread = object). Sync obj when function on obj is accessed by multiple threads.

**OpenMP**: add directives for serial -> ||. Block parallelism {}, #include “omp.h”, gcc -fopenmp -O0-3, **#pragma omp parallel for**: parallel for loop. Put local vars inside loop to stop shared access (*thread private*). **Loop independence**: each iteration does not depend on the previous (*serial equiv.)* **False sharing**: two processes update different elems of same cache line (invalidates cache line on other process, even though there’s no overlap). Best solution: declare on thread’s private stack**. Loop unrolling**: rewrite iterations to save cond. check/i++. X-times unrolling: each unrolled iteration has X iterations. **Loop Fusion**: do work of multiple different loops in one to save on cond. checks/thread overhead once per iteration. **Loop Fission**: Divide work into multiple loops to fit into smaller cache lines. **Separable Dependencies**: compute total using subcalcs from constituents (max of maxes). **Reduction**: || aggregate quantity over loop by computing partial results and combining at thread end. **Reducer**: allows multiple threads to contribute to one value thru identity (default val) and combine (put two vals together)

**Global Interpreter Lock (GIL)** prevents running two ops in ||. Use **JobLib** in || Python. “**Delayed**”: delay exec of f() to later. “**Parallel**”: create ind. workers to do work of f() in parallel. **Concurrent**: (threads) multiple async actions. **Parallel**: (processes) simultaneous exec. Concurrent just means being worked towards at the same time as other things, parallel means equal work done in equal time. Processes each have their own GIL, threads share one GIL.

**Multicore**: each core has own cache/processing power, shared memory. **Threads** run on one core; **processes** run on multiple cores. **Moore’s Law**: (ended?) # transistors for CPU doubles every 2 years. **Dennard Scaling**: (-2006) smaller transistors -> constant power so faster clock rates. Memory Hierarchy: L0 (registers), L1 (cache for core), L2 (on-chip shared), L3 (off-chip cache), Main memory (DRAM – pages from disk), local storage (disks), remote storage (web…). Performance cost. **Access**: using all data from line at once? Aligned: access range starts/ends on cache bound., **Seq**.: continuous range of bytes, **Coalesced**: combine multiple smaller -> one big. **Strided**: one elem used for every column of data (BAD). Fastest varying elem should be innermost.

**Speedup**: T(1)/T(n). **Amdahl’s**: , or

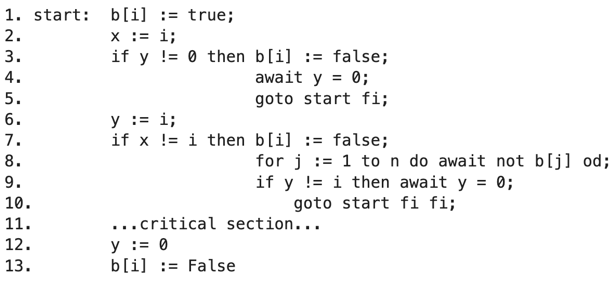
p is parallel, s is resources, S is speedup of whole. Optimal when p=1, S=s (fully optimized code where doubling resources doubles speed). Speedup plot: log scale, speedup vs resources. Can compute optimal speedup by changing # resources and measuring speedup and solving for p -> use 1/(1-p) for optimal. Useful for choosing whether to parallelize/what parts to optimize

**Fork**: set off parallel thread; **Join**: wait till all threads are done. **Span**: Longest chain of dependent ops (time to compute with infinite resources). Parallelism = # ops / span. **Cilk**: fork/join in C/C++. Simpler and dynamic scheduling, but less expressive. *Cilk\_spawn,* *Cilk\_scope*, *Cilk\_for*

**Contention**: two threads accessing shared resources. **Cooperation**: one process action enables another. **Deadlock**: when all threads are waiting on shared resource. **Mutual Exclusion**: *exclusive access to shared resource*, *no deadlock*, ***starvation resistant*** (all threads must make progress eventually). **Peterson’s** is mutually exclusive, starvation resistant, contention-free overhead = 4, but only for two parties, with busy waiting in await (good for low contention resources + short wait, bad otherwise – sleep/restart is alternative, with more overhead to start/stop, but free processors during). **Bakery**: take a ticket at the counter and wait your turn, with thread # as priority for ties. Fast Mutex is mutually exclusive, deadlock free, contention-free overhead = 7, but starvation is possible.

**Total Possible FLOPS/s** is bounded by either: *process rate* = off-chip BW \* operational intensity (# times you use each byte = # FLOPS/(GB/s)) or *peak processing thruput.* Minimum of these two is the **roofline** (low intensity -> process rate wins -> memory bound, higher intensity -> peak thruput wins -> compute bound). No possible program can exceed roofline. Multicore processing makes it worse (all memory bound programs perform the same, some CPU-bound programs become memory bound, only the densest programs see benefit). Use Roofline to determine what to improve (enhance CPU peak, change operational intensity, improve data rate to shift memory bound). Historical trend is roofline moves up/right – bad.

**ILP**: || execution of sequence of instructions in program (# inst. / cycle). **Vector processing:** single instruction multiple data (SIMD), usually done auto by compiler. Operations on long contiguous memory and accumulate results. **Intrinsics**: call to vector instructions in C as functions – only compile for one architecture, so usually have base and base + intrinsics. **Instruction Pipelining**: overlapping independent instructions. **Out-of-order exec**: reorder instructions for better resource use. **Branch predict**: guess outcome of if-else and pad pipeline with next instructions

A diagram of a algorithm

Description automatically generatedA screenshot of a computer program

Description automatically generated