Efficient Use of Multicore Processors For Timing Analysis

Dr João Geada

Contents

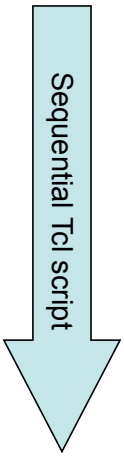
- A brief overview of goals
- Parallelization patterns and applicability
  - Pipelining
  - Latency hiding
  - Dataflow
- Anti-patterns
- The devil in the details
- Final thoughts

Brief Overview of Goals

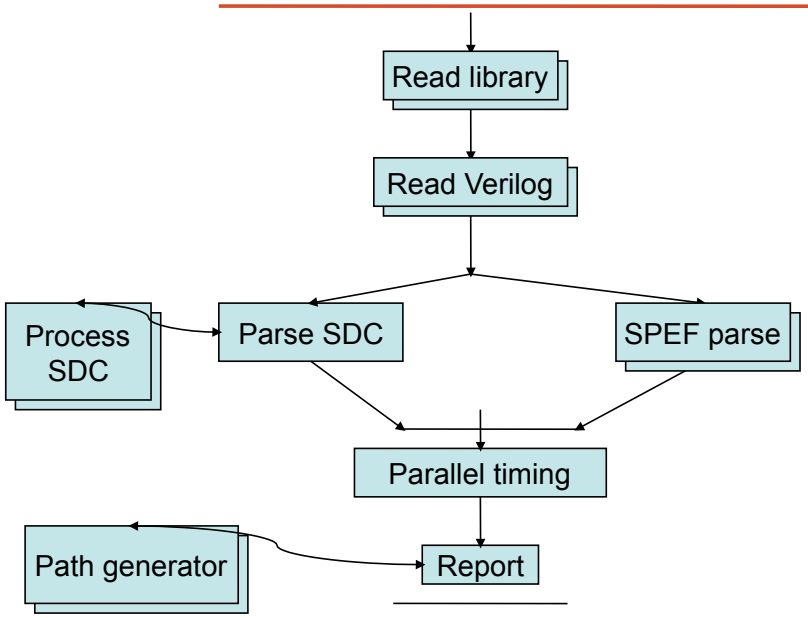
- What do we mean by efficient?
  - Wall clock time !!!
- Goal is to make a timing run take as little wall clock time as possible
  - Multi threads, multi core
  - Acceptable to sacrifice some CPU time for gains in wall clock
- Minimal impact to memory
- Results must always be predictable and repeatable
  - For any timing mode:
    - STA, SSTA, SI, non-SI, transistor, reports
    - Flat, hierarchical, full, incremental
  - For any run environment:
    - Number of CPUs, threads, order of execution

Traditional timing tool

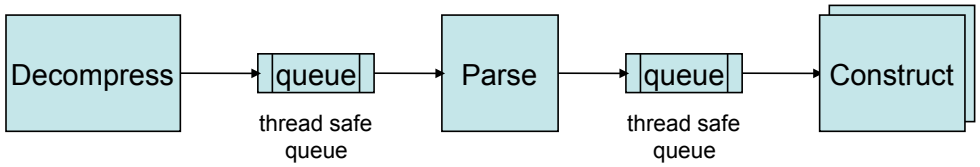
- System setup
  - Read libraries
  - Read design
  - Read parasitics
  - Read and apply SDC
- Evaluate timing model
  - Compute arrivals, constraints, violations
- Report results
  - Report paths



Multicore timing tool



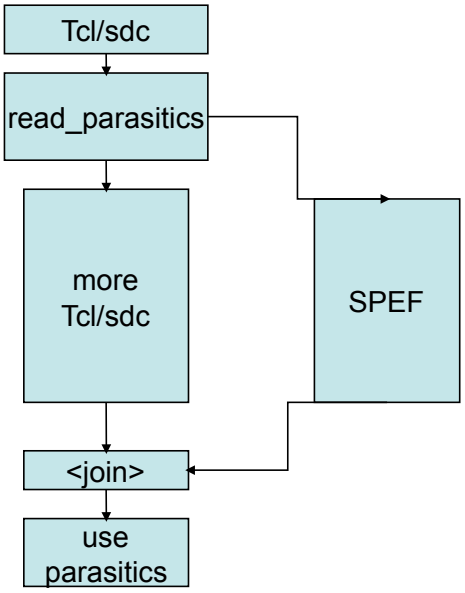
Pipelining



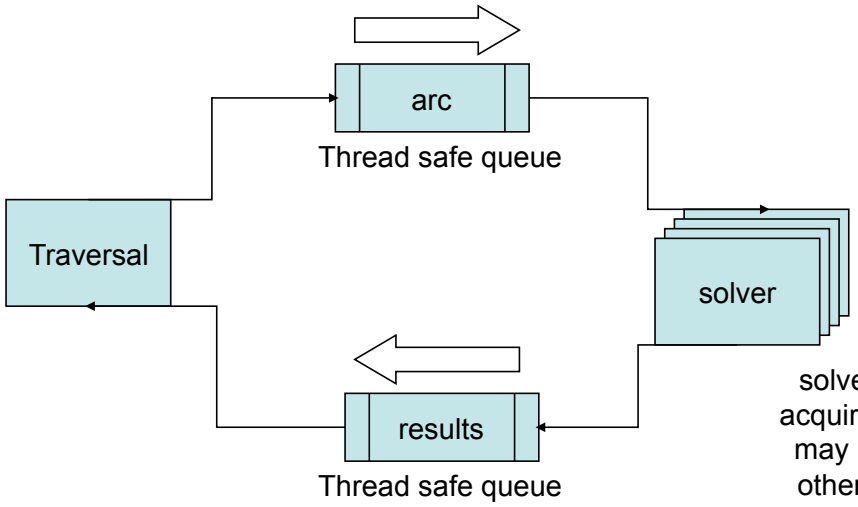
Pattern applicable to parsers: each stage is inherently sequential, but does not need to wait for completion of succeeding/preceding stage. Can resource balance by adding more threads to slow stages.

Latency Hiding

Pattern applicable whenever multiple tasks are done sequentially but output of some of the tasks is not normally observable until later.



Dataflow



solvers cannot acquire locks that may be held by other solvers!!!

Pattern applicable to graph traversals

AntiPatterns

- Partitioning
  - Generally
    easy
    t
    o do, good performance for small number of additional cores
  - Difficult to do for large
    num
    b
    er of cores, highly coupled problems (for example SI timing)
  - Partitioning overhead ultimately affects scaling
- Using multicore aware math libraries
  - Affects too little of the runtime
  - Negatively impacts other
    more
    eff

Lessons learned

- Even a single misplaced instruction can kill performance
  - One instruction accounted for 30% performance!
- Locks are deadly to performance
  - But essential for correctness when needed
- Correct architecture is vital
  - Minimize required locks
  - Guarantee data properties

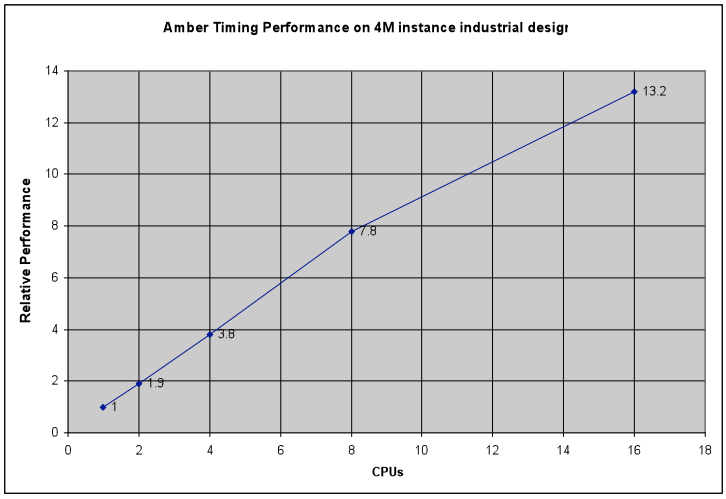
Lessons learned

- Engineers are not trained to use threads
  - Take us about 3 months to bring a new hire up to speed
  - Need to take a very disciplined approach to coding
- State of thread oriented software tools is immature
  - Few tools in general for threading
    - Performance
    - Memory behavior and correctness
    - Data debugging
  - Few threaded building blocks available
- For now, you *will* have to construct your own tools and components to extract the most from multicore CPUs

Final Thoughts

- Amdhal's Law is not your friend
  - On an average timing run Amber spends more time reading input files than running timing
  - As soon as you effectively thread something, the 4-16x performance boost makes something else the critical problem
- Input file formats used in timing not easily amenable to parallelized IO
  - Standard formats are serial in nature (liberty, SPEF, SDC, ...)
  - As an industry we need to start paying more attention to this
  - Tackling IO by providing thread aware incremental database allows for multiple order of magnitude speed ups
- Existing threading primitives are very primitive, and making

Why do multicore/threading?



8 way scaling advantage