Operating Systems, Concurrency and Time

hardware evolution and concurrency

Johan Lukkien
Questions

• Which advances in hardware are relevant and what is their effect on software?
• How to exploit platform concurrency?
Timing and performance requirements

- Time (deadline) constraints imply the need:
  - to perform computations (tasks) sufficiently fast
  - to have a predictable system

- High performance (operations/sec)
  - not required for most (control oriented) real-time applications
    - relatively simple computations
  - is important for compute-bound applications
    - e.g. (media) processing pipelines
    - advanced processing-in-the-(real-time)-loop

- Both real-time and performance requirements lead to parallelism and concurrency
  - pre-emption and scheduling policies of tasks
  - tasks being distributed over several processes / threads / processors
  - introduction of parallelism to achieve sufficient performance to make it in time
Evolution: the only certainty is change
Hardware and Software

- Hardware evolves
  - Moore’s law: ‘faster is better’ (also ‘less energy is better’)
    - further integration
    - new standards
  - exploiting concurrency techniques
  - … challenges for software designers to take proper advantage

- Software evolves
  - to accommodate new requirements, new functionality
    - enabled by advances in hardware
    - requested by end-users, market position
  - also caused by independence in life cycles
    - OS changes
    - concurrency in hardware
    - used packages and tools
Multicore

• Independent, full-fledged CPUs in same IC
  – e.g. ARM Cortex A9 (4),
    Intel I7 5960x (8),
    Intel Xeon E5 2699v4 (22),
    Samsung Exynos (4+4)
  – heterogeneous / homogeneous

• Sharing memory
  – typically, private (L1) cache, shared >L1 caches
  – implementations with memory controller per CPU
    • memory of another CPU is addressed via forms of message
      passing through an interconnect
    • leading to NUMA (non-uniform memory access)
  – advanced cache coherency protocols

• A network topology for interconnect
  – e.g. crossbar, mesh, ring, point-to-point
  – connecting processors to memory (UMA) or connecting P/M pairs (NUMA)

• Multicore fits thread level parallelism
- NUMA across chips
- UMA on chip
ccNUMA

- (cc)(N)UMA: (cache coherent) (non-)uniform memory access
  - physical memory access times dependent on address (*spatial* differences)
  - cache coherency is vital for usability
- Complex cache hierarchies already lead to *temporal* differences
  - and cache coherency is more costly for this NUMA organization

- Handles in the OS: combination of
  - processor affinity for threads
  - explicit memory placement
  - (explicit cache management)

https://software.intel.com/en-us/articles/optimizing-applications-for-numa
Hyperthreading

- Hyper-threading (as off 2001, Intel):
  - the processor supports fetching multiple instruction streams
    - even from different address spaces
  - these streams share internal processor resources
    - e.g. ALU, cache, SIMD resources, fetch/decode
  - a blend of these instructions is executed
    - simultaneous/symmetric: issue an instruction of each thread, each time
    - temporal: interleave instructions of different threads
      - fine grained: alternate
      - course grained: run until blocking
  - Logically, this looks like multiple distinct processors (4 in the example)
    - multiple pipelines/sets of registers
  - Effectively, 30%-50% speedup is achieved with two threads
    - implying that the processor was under-utilized when used without
Stalling

Pipeline stall occurs here because instruction #1 is attempting to store a value to memory at the same time instruction #2 is attempting to read a value from memory.

Instruction #3 appears to take two clock cycles to execute because of the pipeline stall.
Using hyperthreading

- How to predict whether it is advantageous?
  - how is execution time predictability?
  - increase in average performance

- How to control?

- Experiment!
  - A low priority thread might delay a high priority thread when mapped on the same physical processor
    - this happens as they are still different processors logically
    - there is no control over resource sharing policy

- Mainly under OS control
  - understand OS scheduling policies
  - processor affinity techniques
    - but interference with ‘background’ processes remains
• Single Instruction, Multiple Data
  – same operation on large number of data
  – algorithms: graphics, linear algebra

• Moved into Intel x86 instruction set extensions
  – MMX, multimedia extensions
  – SSE...2,3,4, streaming SIMD extensions (1999+)
  – AVX, AVX2, AVX512 (3.1,3.2), advanced vector extensions (2015)
    • extensive: e.g. dot-product

• SIMD is a processing resource shared by hyperthreads

• Typical use: standard library, fixed API, evolving implementation
  – SIMD library
  – Intel Integrated Performance Primitives (IPP)
  – Intel Math Kernel Library (incl. multi-threading)
Summary of changes

OS (+ compiler!)
- More parameters, more settings, different defaults
- Size (memory footprint) matters!
- Differences in resource management
  - scheduler, scheduling
  - memory (de-)allocation
- New services
- Many silent assumptions
- … different timing and execution traces

Hardware
- Increased concurrency (often implicit)
  - execution pipelines, depth
  - speculative execution
  - SIMD, MIMD
- Changes in interconnect
  - QPI, NUMA
- Changes in memory organization
  - caches
- … different OS decisions
- … different timing and execution traces

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
18-Sep-16
Examples of transition problems

- Superfetch (and other intelligence):
  - loads frequently used software upon startup
  - but impacts the entire startup period and disk access during operation
- Scheduling:
  - mapping decisions for (compute intensive) tasks
    - domain decomposition requires co-scheduling for low latency
- Memory footprint:
  - increase in OS size and/or common services
    - leading to the virtual memory system (swapping) kicking in
- System call latency:
  - system calls in time critical tasks lead to dependencies on those system calls, and everything these depend upon
  - time critical tasks should use asynchronous communication (IO)
How to introduce concurrency in applications?
Techniques for parallelism for speed

- **functional parallelism**: decompose the function to be computed
  - pipelining:
    - e.g. instruction pipelining, graphics pipeline
    - increases throughput, (slightly) higher latency
  - divide and conquer: divide the problem (in two or more parts) and compute independently
    - e.g. merge sort,
    - needs ‘split’ and ‘combine’
    - decreases latency

- **data parallelism**: decompose the data and process the same function in parallel on parts of the data
  - e.g. computing on a 2D image, matrix operations, 2D FFT

- **result parallelism**: execute the same function, in parallel on many distinct data items at the same time
Example: MapReduce (Google, Apache)

- **Map:**
  - Accepts *input* key/value pairs
  - Emits *intermediate* key/value pairs
- **Partition:**
  - map intermediate key range to the set of $R$ Reducers
    - e.g. $(k, v) \rightarrow \text{hash}(k) \mod R$
- **Sort:**
  - identical keys are taken together and their values are put in a single list; keys are sorted
- **Reduce:**
  - Accepts an *intermediate* key/(value-sequence) pair
  - Emits *output* key/value pair

Figure: data flow in MapReduce
Example: mapReduce for word count

• map:
  – generates list of (word, “1”) pairs

• reduce:
  – sum the values per key

• Programming model:
  – three (+ ..) API calls
    • map(), reduce(), compare()
  – underlying file system support
  – further tools to manage this

map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));

MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat,
OSDI’04: Sixth Symposium on Operating System Design and Implementation,

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
Example: matrix multiplication

\[
\text{for } i := 0 \text{ to } N-1 \text{ do} \\
\text{for } j := 0 \text{ to } N-1 \text{ do} \\
C[i,j] := \text{DotProduct}(A[i,*], B[*j])
\]

- Each process(or) performs the computations required for the part of C it is assigned to.

- To that end it may need to communicate locally stored parts of A and B to other processors, and receive parts of A and B it does not have.

- On an OS with an adaptive memory allocation strategy this happens automatically; otherwise, explicit placement might help.

- Co-scheduling is required in order to achieve reduction of latency
Possible data distributions

- striped
- blocked
- scattered (cyclic)

**Question:**
- which distribution is good (communication-wise) for
  - matrix multiplication
  - matrix transposition
- can you use MapReduce for this problem as well?
  - what’s the difference?
Combining

• The three techniques may be applied together
  – e.g. computing the partial sums in a matrix multiplication in a pipelined or treelike fashion using SIMD
  – performing many different matrix multiplications as result parallelism

• Different *granularity* levels should be balanced
  – e.g. if result parallelism is possible and single job latency is not the problem .... use it!
  – avoid communication ... it is overhead
    • hence, parallelism at highest possible level (largest grain size)
    • (extensive) distribution strategies should not take more time than the original computation...

• It may be possible that an inferior sequential algorithm is more amenable to parallel execution
  – e.g. Gauss-Jordan elimination
Exercise

• Examine your own application, or one in Philips you are familiar with.
  – which concurrency in hardware is exploited?
  – are concurrency techniques (like functional or data parallelism) useful?
  – describe a problem you experienced because of a transition to a new environment.