Sprachkonzepte der Parallelen Programmierung Vorlesung SS 11, WS 12, SS 13, WS 15–17

Johannes Waldmann, HTWK Leipzig

11. Oktober 2017

Sprachkonzepte der parallelen Programmierung

- programming language concepts
  for concurrent, distributed, and parallel computing
- why? 1. application requires it, 2. hardware allows it
- optional course for BSc. computer science students, in their 5th semester (of 6)
- each week (of 14): one lecture, one lab class (discussing homework, programming exercises)
- finally, written exams (closed book) 120 min

Concepts of Parallelism

- non-deterministic
  - concurrent (nebenläufig)
    interleaved execution of components
  - distributed (verteilt)
    as above, plus: explicit data transfer (messages)

  e.g., responsive multi-user system requires concurrency
  (can be simulated on sequential hardware: OS)
- deterministic
  for application probl. without concurrency requirement,
  use hardware parallelism to solve them faster
  e.g., matrix multiplication, text/image analysis

From Concept to Implementation

notice the gap:

- quite often, we want deterministic parallelism
- but hardware is concurrent (e.g., several cores) and distributed (e.g., per-core memory/cache)

who bridges the gap?

- WRONG: the *programmer* (application program handles sequencing, synchronisation and messaging)
- RIGHT: the *language* (with libraries, compiler, run-time system) (program expresses intent)

note the difference between: $\sum_{0 \le i < n} x_i$ (intent)
and: `for(i=0;i<n;i++){s+=x[i];}` (sequencing)

Abstract! Abstract! Abstract!

main thesis

- *higher* abstraction level of the language
- $\Rightarrow$ *easier* for compiler and RTS to use hardware specifics (e.g., parallelism) for efficient execution

example (C#, mono) just one annotation expresses the intent of parallel execution:

```
Enumerable.Range(0,1<<25)
      .Select(bitcount).Sum()
Enumerable.Range(0,1<<25).AsParallel()
      .Select(bitcount).Sum()
```

this is why we focus on functional programming
(e.g., `Select` is a higher-order function)

Why did this work, exactly?

`Enumerable.Range(...).AsParallel().Sum()`

- technically, `AsParallel()` produces `ParallelQuery<T>` from `IEnumerable<T>`, and `Sum()` has a clever implementation for that type
- mathematically ("clever"), addition is *associative*, so we can group partial sums as needed
- if an operation is not associative?
  e.g., the final carry bit in addition of bitvectors
  then we should find a modification that is!
  because that allows for *straightforward*
  and *adaptable* (e.g., to number of cores) parallelism

Types for Pure Computations

measure run-times and explain (C# vs. Haskell)

```
Enumerable.Range(0,1<<25).Select(bitcount).Sum()
Enumerable.Range(0,1<<25).Select(bitcount).Count()
Enumerable.Range(0,1<<25)                  .Count()
length $ map bitcount $ take (2^25) [ 0 .. ]
length                $ take (2^25) [ 0 .. ]
```

- elements of list are not needed for counting
- computation of elements cannot be observed
  it has no side effects (Nebenwirkung)
  (this follows from `bitcount:: Int -> Int`)
  the Haskell RTS never calls `bitcount`,
- the C# type `int->int` includes side effects,
  so the RTS must call the function.

If we absolutely must program imperatively,

(imp. program execution $=$ sequence of state changes)

- then we are on dangerous ground
  already for the sequential case
  proving an imperative program correct requires
  complicated machinery (Hoare calculus)
  hence it is often not done
  (for functional programs just use equational reasoning)
- we need even more caution (and discipline)
  for concurrent imperative programs
  need concurrency primitives
  - that have clear semantics
  - that solve typical problems
  - that are composable (to larger programs)

Typical Concurrency Problems

- mutual exclusion
  at most one process gets to access a shared resource
  (e.g., a shared memory location)
- producers and consumers, readers and writers
  - cannot consume item before it is produced,
  - cannot consume item twice
- concurrent mutable data structures
  - counters
  - collections (hash maps, . . .)

Semantics for Concurrent Systems

. . . via mathematical models:

- Petri nets (Carl Adam Petri, 1926–2010)
  (automata with distributed state)
- process algebra (Bergstra and Klop 1982, Hoare 1985)
  (regular process expressions and rewrite rules)
  `http://theory.stanford.edu/~rvg/process.html`
- modal logic
  (statements about time-dependent properties)
  application: model checking,
  e.g., SPIN `http://spinroot.com/`

Concurrency Primitives

- locks (Semaphores, E.W. Dijkstra 1974) `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF`
  of historical importance, but … *locks are bad*
  (in particular, not composable)
- no locks
  atomic, non-blocking ("optimistic") execution of
  - elementary operations (compare-and-swap)
    realized in hardware
    `http://stackoverflow.com/questions/151783/`
  - transactions (STM, software transactional memory)
    `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/` in Haskell, Clojure

Homework

1. which are associative? (give proof or counter-example)
    1.1 on $\mathbb{Z}$: multiplication, subtraction, exponentiation
    1.2 on $\mathbb{B}$ (booleans): equivalence, antivalence, implication
    1.3 on $\mathbb{N}^2$: $(a, b) \circ (c, d) := (a \cdot c, a \cdot d + b)$
2. sum-of-bitcounts
    2.1 re-do the C# bitcounting example in Java (hint: `java.util.stream`)
    2.2 discuss efficient implementation of `int bitcount (int x);` (hint: time/space trade-off)
    2.3 discuss efficient implementation of sum-of-bitcounts
        2.3.1 from 0 to $2^e - 1$
        2.3.2 bonus: from 0 to $n - 1$ (arbitrary n)

    hint:
    how did little Carl Friedrich Gauß do the addition?
    morale:
    the computation in the example should never be done in real life, but it makes a perfect test-case since it keeps the CPU busy and we easily know the result.
3. sorting network exercise: `https://autotool.imn.htwk-leipzig.de/new/aufgabe/2453`
4. on your computer, install compilers/RTS for languages: