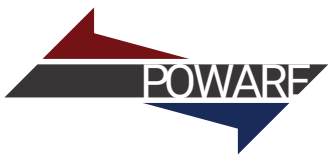
Multithreading with Tasklets

Andreas Lauser

January 25, 2019



Overview

Overview

Sequential Computing: (Deterministic) Turing Machine

- Model: Mathematician
- Brain with finite amount of memory
- Pencil, Rubber & Infinite amount of scratch Paper
- Capacity to run simple Instructions
- List of Instructions

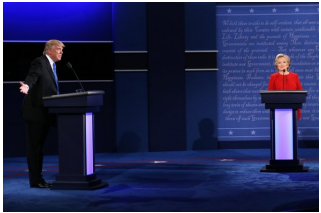
Sequential Computing: (Deterministic) Turing Machine

- Model: Mathematician
- Brain with finite amount of memory
- Pencil, Rubber & Infinite amount of scratch Paper
- Capacity to run simple Instructions
- List of Instructions

$$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$$

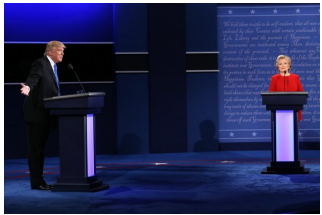
Q	Finite sets of states
Γ	Alphabet
b	Blank character
$\Sigma \subseteq \Gamma \setminus \{b\}$	Input symbols
q_0	Initial state
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$	Transition relation
$F \subseteq Q$	Final states

Parallel Computing



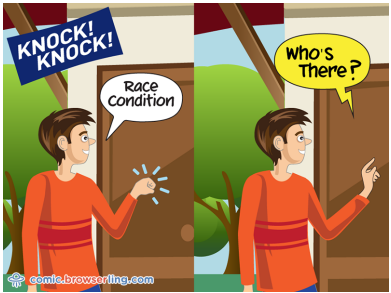
- Model: Group of Mathematicians
- Can talk to each other
- Prefer to work on their own
  - Communication is relatively inefficient
- Either each gets an own sheet of paper or all work on the same sheet

Processes and Threads



- Process: Each mathematician gets a separate pile of scratch paper (dedicated main memory region)
- Thread: All mathematicians edit the same pile of scratch paper (shared main memory region)
- Process model generally requires more explicit (messages) but less implicit communication (locks, semaphores, etc.) for synchronization
  - Generally assumed to be slower than doing the same work using the threaded approach
  - Grain of salt: Milage may vary; modern CPUs are complex beasts
- Threading model quickly leads to conflicting edits of the shared paper (race conditions)
  - Typically easily to do and very hard to find

Race Conditions



- Sometimes the outcome of the computation depends on the precise timing of computation
- Consider two threads both running $a = 1; a = a + 2$
- Possible results: $a = 3$ and $a = 5$
- Solution: Introduce critical sections via synchronization primitives
  - Lock/Mutex: The first applicant gets in
  - Semaphore: A given number of threads may enter the critical section
  - Condition Variables: A critical section can only be entered if a given predicate is fulfilled

Overview

Tasklets

- Tasklet: Work that can be deferred and run in a different thread
  - Implemented on top of C++-2011 primitives
  - Specified via simple callback function/lambda ...
  - ... or object derived from `Ewoms::TaskletInterface` which can be used to hold additional scaffolding
- Thread pool: Multiple “day laborer” threads queueing to work on the next tasklet. Work is distributed by the `Ewoms::TaskletRunner` class.
- Barrier: All tasklets must be finished before continuing

Tasklets Example

```
Ewoms::TaskletRunner runner(/*numWorkers=*/3);

auto fn =
  [&runner]()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    std::cout << "I am worker " << runner.workerThreadIndex() << std::endl;
};
for (int i = 0; i < 5; ++ i)
    runner.dispatchFunction(fn);
runner.barrier();
std::cout << "I am the main thread" << std::endl;
```

- Mechanism to easily parallelize “most” workloads
- Available for C/C++ and FORTRAN
- Requires compiler support; needs to be explictly enabled using compiler flags
- Does not take advantage of “modern” C++ >= 2011 constructs
- Well established:
  - First version of the standard is from 1997
  - Supported by all common compilers: GCC, Clang, VC++, ICC, . . .

OpenMP Example

```
omp_set_num_threads(/*numWorkers=*/3);

#pragma omp parallel for
for (int i = 0; i < 5; ++ i) {
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    std::cout << "I am worker " << omp_get_thread_num() << std::endl;
};

// barrier is implicit (mandatory?). explicit: #pragma omp barrier

std::cout << "I am the main thread" << std::endl;
```

Tasklets vs OpenMP

OpenMP:

- Direct parallelization of “simple” loops
- Build-in schedules (static, dynamic, . . .)
- Established
- Requires special compiler support and explicit enabling
- Single thread pool

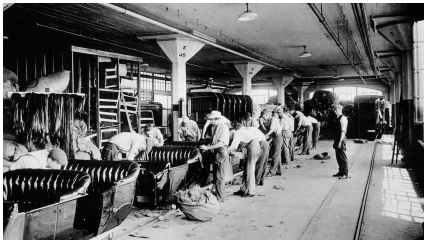
Tasklets:

- No special compiler support required: Build on top of standard C++-2011 primitives
- All synchronization between tasklets has to be explicit
- Arbitrary number of thread pools easy to implement
- More flexibility w.r.t. task handling (e.g., setting CPU affinity)

Asynchronous Output

- Writing to harddisk is slow
  - In particular on some network file systems and on Windows
- Computer can do useful work while waiting for I/O to finish
- Approach: Extract data to be written in main thread, hand off the writing to a tasklet in a dedicated thread pool
- Currently VTK and ECL data can be written asynchronously

Multi-Threaded Linearization



Approach:

- Only linearization thread-parallelized: Problems with ordering dependent preconditioners for linear solvers
- Each thread loops over the grid, skips the elements which are taken by other threads
  - `Ewoms::ThreadedElementIterator`
- Linearize the localized solution for the current element
- Update the global residual and its Jacobian using the local linearization. Use critical sections to avoid race conditions when accessing global objects
- Currently implemented using OpenMP
- OpenMP used for historical reasons
  - Feels a bit shoehorned
  - Patch to transition to tasklets available as pull request

Summary

- Process-level and Thread-level parallelization
- Thread-level parallelization requires less message passing, but quickly leads to race conditions
- On multi-core desktop-class computers, thread-level parallelization can be made at least as performant as process-level parallelization
- In OPM, output writing and linearization is multi-threaded
  - Tasklets used for asynchronos output writing and OpenMP for linearization
- Linear solvers not easily thread-parallelizable
  - Ordering dependence of practically relevant preconditioners

Outlook

- Single mechanism for thread-parallelization is desireable
- Multiple independent thread-pools required, i.e., OpenMP not feasible

That’s it. Questions?!