Lecture 18: Abstract vs Real Performance - An “under the hood” look at HJlib

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
HJ-lib Compilation and Execution Environment

Java 8 IDE → javac Foo.java → HJ-lib source program is a standard Java 8 program

Foo.java → Java compiler → Foo.class

Java compiler translates Foo.java to Foo.class, along with calls to HJ-lib with lambda parameters (async, finish, future, etc)

Java Foo → HJ runtime initializes m worker threads (value of m depends on options or default value)

HJ runtimes + HJ-lib libraries → HJ Abstract Performance Metrics (enabled by appropriate options)

HJ-lib Program Output

All the "magic" happens here!
Looking under the hood - let’s start with the hardware

Main Memory (DRAM)
How does a process run on a single core?

Processes are managed by OS kernel
- Important: the kernel is not a separate process, but rather runs as part of some user process

Control flow passes from one process to another via a context switch

Context switches between two processes can be very expensive!

Source: COMP 321 lecture on Exceptional Control Flow (Alan Cox)
What happens when executing a Java program

• A Java program executes in a single Java Virtual Machine (JVM) process with multiple threads

• Threads associated with a single process can share the same data

• Java main program starts with a single thread (T1), but can create additional threads (T2, T3, T4, T5) via library calls

• Java threads may execute concurrently on different cores, or may be context-switched on the same core

Figure source: COMP 321 lecture on Concurrency (Alan Cox)
Thread-level Context Switching on the same processor core

- Thread context switch is cheaper than a process context switch, but is still expensive (just not “very” expensive!)
- It would be ideal to just execute one thread per core (or hardware thread context) to avoid context switches

Figure source: COMP 321 lecture on Concurrency (Alan Cox)
Now, what happens is a task-parallel Java program (e.g., HJ-lib, Java Fork, etc.)

- HJ-lib runtime creates a *small number of worker threads*, typically one per core

- Workers push new tasks and “continuations” into a logical work queue

- Workers pull task/continuation work items from logical work queue when they are idle (remember greedy scheduling?)
Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core

Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core
- And of customers as tasks

Image sources: http://www.deviantart.com/art/Randomness-20-178737664,
http://www.wholefoodsmarket.com/blog/whole-story/new-haight-ashbury-store
All is well until a task blocks ...

- A blocked task/customer can hold up the entire line
- What happens if each checkout counter has a blocked customer?

source: http://viper-x27.deviantart.com/art/Checkout-Lane-Guest-Comic-161795346
Approach 1: Create more worker threads (as in HJ-Lib’s Blocking Runtime)

- Creating too many worker threads can exhaust system resources (OutOfMemoryError)
- Leads to context-switch overheads when blocked worker threads get unblocked

source: http://www.deviantart.com/art/Randomness-5-90424754
• Assume that five tasks (A1 … A5) are registered on a barrier.
• Q: What happens if four tasks (say, A1 … A4) executing on workers w1 … w4 all block at the same barrier?
• Assume that five tasks (A1 … A5) are registered on a barrier

• Q: What happens if four tasks (say, A1 … A4) executing on workers w1 … w4 all block at the same barrier?

• A: Deadlock! (All four tasks will wait for task A5 to enter the barrier.)

• Blocking Runtime’s solution to avoid deadlock: keep task blocked on worker thread, and create a new worker thread when task blocks
Blocking Runtime (contd)

- Examples of blocking operations
  - End of finish
  - Future get
  - Barrier next

- Approach: Block underlying worker thread when task performs a blocking operation, and launch an additional worker thread.

- Too many blocking operations can result in exceptions and/or poor performance, e.g.,
  - `java.lang.IllegalStateException: Error in executing blocked code! [89 blocked threads]`
  - Maximum number of worker threads can be configured if needed.
    - `HjSystemProperty.maxThreads.set(100);`
Approach 2: Suspend task continuations at blocking points (as in HJ-Lib’s Cooperative Runtime)

- Upon a blocking operation, the currently executing tasks suspends itself and yields control back to the worker
- Task’s *continuation* is stored in the suspended queue and added back into the ready queue when it is unblocked
- Pro: No overhead of creating additional worker threads
- Con: Need to create continuations (enabled by -javaagent option)
Continuations

• A continuation can be a point immediately following a *blocking* operation, such as an end-
  `finish`, `future get()`, `barrier/phaser next()`, etc.

• Continuations are also referred to as task-switching points
  —Program points at which a worker may switch execution between different tasks (depends on
  scheduling policy)

```cpp
   \text{1.} \text{finish \{ // F1}
   \text{2.} \text{async A1;}
   \text{3.} \text{finish \{ // F2}
   \text{4.} \text{async A3;}
   \text{5.} \text{async A4;}
   \text{6.} \}
   \text{7.} S5;
   \text{8.}\}
```

Continuations
Cooperative Scheduling (view from a single worker)

Cooperative runtime automatically creates continuations at suspend points via

Useful work for some other task on same worker thread

Task-1
suspend
resume
suspend

Task-2
suspend/completed
Any operation that contributes to unblocking a task can be viewed as an event e.g., task termination in finish, return from a future, signal on barrier, put on a data-driven-future, …
Why are Data-Driven Tasks (DDTs) more efficient than Futures?

- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all Data-Driven Futures (DDFs) are available.
  - An “asyncAwait” statement does not block the worker, unlike a future.get()
  - No need to create a continuation for asyncAwait; a data-driven task is directly placed on the Suspended queue by default.

- Therefore, DDTs can be executed on a Blocking Runtime without the need to create additional worker threads, or on a Cooperative Runtime without the need to create continuations.
Summary: Abstract vs Real Performance in HJ-Lib

- **Abstract Performance**
  - Abstract metrics focus on operation counts for WORK and CPL, regardless of actual execution time

- **Real Performance**
  - HJlib uses ForkJoinPool implementation of Java Executor interface with Blocking or Cooperative Runtime (default)
Announcements & Reminders

- HW3 CP1 is due Friday, Feb 28th at 11:59pm
- Watch the topic 5.1, 5.2, 5.6 videos for the next lecture
- Midterm exam (Exam 1) will be held at 7pm on Thursday, February 27, 2020 in Duncan Hall McMurtry Auditorium
  - Closed-notes, closed-book exam scheduled for 2 hours during 7pm — 9pm (but you can leave early if you’re done early!)
  - The exam will be in Canvas. You are allowed to use your laptop ONLY to enter your answers in Canvas, nothing else