Thread Libraries & Scala Agents

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 19 — 03/16/2010

Goals

- Threading Libraries
  - Review material from Chapter 5 of Breshears
  - Implicit Threading
    - OpenMP, Intel Threading Building Blocks, Scala Agent Model, go's goroutines, Clojure's concurrency constructs...
  - Explicit Threading
    - Pthreads, Windows Threads, JDK, ruby, python, etc.
- Introduce the Scala Agent Model

Goals

- Threading Libraries
  - Review material from Chapter 5 of Breshears
  - Implicit threading libraries manage threads for you
    - OpenMP, Intel Threading Building Blocks, …
  - Explicit Threading
    - Pthreads, Windows Threads, …
- Introduce the Scala Agent Model

Implicit Threading

- Implicit threading libraries handle the task of
  - creating,
  - managing, and
  - synchronizing threads
- If your concurrency needs can be handled by the limited features of an implicit threading library then
  - you can write concurrent programs and not bother with the details of thread management
- We will only show high-level examples in this lecture

Examples

- OpenMP
  - A set of compiler directives, library routines and environment variables that specify shared-memory concurrency in FORTRAN, C, and C++
- Intel Threading Building Blocks
  - A C++ template-based library for loop-level parallelism that concentrates on defining tasks rather than explicit threads
- Many others
  - e.g. Scala agent model, go’s goroutines, Clojure’s refs, atoms & agents, etc.

OpenMP (I)

- OpenMP directives indicate code that can be executed in parallel; such sections are called parallel regions
  - These directives control how code is assigned to threads
  - To define a parallel region in C++, use a pragma
    - #pragma omp parallel
  - This pragma will be followed by a block of code (or even a single statement) which will be assigned to threads automatically, executed in parallel and automatically joined back to the main thread of control

OpenMP (II)

- The omp for construct will make a for loop concurrent
  - There are options for static and dynamic scheduling
  - There are options to make statements atomic or to ensure that only a single thread executes a statement'
  - There are options for specifying reductions (combining a set of values across multiple threads into a single value)
  - Finally, OpenMP provides features for creating thread-local storage:
    - for instance, loop variables are made thread specific, as are any variables declared inside a parallel region

Returning to Pi

- We return to the multithreaded program we saw earlier this semester that calculates an approximate value of PI
- The example code demonstrates an OpenMP parallel region with thread local storage and an automatic reduction
- In particular
  - `#pragma omp parallel for private(mid, height) reduction(+:sum)`
- parallel section, parallel for loop
- private vars mid and height
- automatic reduction of sum variable across threads using the plus operator, storing result in sum

```
static long num_rects = 1000000;

int main(int argc, char* argv[]) {
    double mid, height, width, sum = 0.0;
    int i;
    double area;

    width = 1.0/(double)num_rects;

    #pragma omp parallel for private(mid, height)
    reduction(+:sum)
        for (i = 0; i < num_rects; i++) {
            mid = (i + 0.5) * width;
            height = 4.0/(1.0*mid*mid);
            sum += height;
        }

    area = width * sum;
    printf("The value of PI is %f\n", area);
    return 0;
}
```

Results?

- Work being assigned to both cores on this machine
  - but utilization never goes over 100% CPU for this process
- Not clear why performance is not higher
  - but, I didn't have to write a single line of code to create threads, assign work to them, worry about sharing values across threads, synchronizing them, etc.

Explicit Threading

- Explicit threading libraries require the programmer to control all aspects of threading, including
  - creating threads
  - assigning tasks
  - synchronizing/controlling interactions between threads
  - managing shared resources

Examples

- Pthreads
  - Stands for POSIX threads, available on a wide number of platforms
- Window Threads
  - Similar library created by Microsoft for Windows platform
- BUT, explicit threading libraries are available in any language in which thread creation/management are the responsibility of the programmer: Java, ruby, python, C#, C, etc.

Pthreads

- Provides basic concurrency primitives to C programs
  - pthread_t is core data structure
  - pthread_create creates new threads
  - pthread_join will join a thread to the main thread of control
  - pthread_mutex_lock provides mutual exclusion
  - pthread_cond_wait() and pthread_cond_signal() provide functionality similar to Java's wait() and notify() methods
- Demonstration

Alternative Approaches

- As a result of these concerns, computer scientists have searched for other ways to exploit concurrency
    - in particular using techniques from functional programming
- Functional programming is an approach to programming language design in which functions are
    - first class values (with the same status as int or string)
        - you can pass functions as arguments, return them from functions and store them in variables
    - and have no side effects
        - they take input and produce output
        - this typically means that they operate on immutable values

Example (I)

- In python, strings are immutable

```
a = “Ken @@@”

b = a.replace(“@”, “!”)

b

'Ken !!!'

a

'Ken @@@'
```

- replace() is a function that takes an immutable value and produces a new immutable value with the desired transformation; it has no side effects

Example (II)

- Functions as values (in python)

```
def Foo(x, y):
  return x + y
add = Foo
add(2, 2)
  → 4
```

- Here, we defined a function, stored it in a variable, and then used the “call syntax” with that variable to invoke the function that it pointed at

Example (III)

- continuing from previous example

```
def DoIt(fun, x, y): return fun(x,y)
```

```
DoIt(add, 2, 2)
```

    - 4

- Here, we defined a function that accepts three values:
    - some other function and two arguments
- We then invoked that function by passing our add function along with two arguments ;
- DoIt() is an example of higher-order functions: functions that take functions as parameters
    - Higher-order functions are a common idiom in functional programming

Relationship to Concurrency?

- How does this relate to concurrency?
    - It offers a new model for designing concurrent systems
        - Each thread operates on immutable data structures using functions with no side effects
        - A thread's data structures are not shared with other threads
        - Work is performed by passing messages between threads
            - If one thread requires data from another that data is copied and then sent
- Such an approach allows each thread to act like a single-threaded program; no danger of interference

Map, Filter, Reduce

- Three common higher order functions are map, filter, reduce
- map(fun, list) -> list
  - Applies fun() to each element of list; returns results in new list
- filter(fun, list) -> list
  - Applies boolean fun() to each element of list; returns new list containing those members of list for which fun() returns True
- reduce(fun, list) -> value
  - Returns a value by applying fun() to successive members of list (total = fun (list[0], list[1]); total = fun(total, list[2]); …)

Examples

- list = [10, 20, 30, 40, 50]

- def double(x): return 2 * x
- def limit(x): return x > 30
- def add(x,y): return x + y

- map(double, list) returns [20, 40, 60, 80, 100]
- filter(limit, list) returns [40, 50]
- reduce(add, list) returns 150

Implications

- map is very powerful
  - especially when you consider that you can pass a list of functions to it and then pass a higher-order function as the function to be applied
    - for example
      - def DoIt(x): return x()
      - map(DoIt, [f(), g(), h(), i(), j(), k()])
- But the real power, with respect to concurrency is that map is simply an abstraction that can, in turn, be implemented in a number of ways

Single Threaded Map

- We could for instance implement map() like this:
  - def map(fun, list):
    - results = []
    - for item in list:
      - results.append(fun(item))
    - return results
- This would implement map in a single threaded fashion

Multi-threaded Map

- We could also implement map like this (pseudocode):
  - def Mapper(Thread):
    - def __init__(… fun, list): …
    - def run():
      - self.results = map(fun, list)
  - def xmap(fun, list):
    - split list into N parts where N = number of cores
    - create N instances of Mapper(fn, list_i)
    - wait for each thread to end (in order) and grab results
    - append thread results to xmap results
    - return xmap results

Note: threads can complete in any order since each computation is independent

Super Powerful Map

- We could also implement map like this:
  - def supermap(fun, list):
    - divide list into N parts where N equals # of machines
    - send list_i to machine i which then invokes xmap
    - wait for results from each machine
    - combine into single list and return
- Given this implementation, you can apply a very complicated function to a very large list and have (potentially) thousands of machines leap into action to compute the answer

Google

- Indeed, this is what Google does when you submit a search query:
  - def aboveThreshold(x): return x > 0.5 <-- just making this up
  - def probabilityDocumentRelatedToSearchTerm(doc): …
  - searchResults =
    - filter(aboveThreshold,
      - map(probabilityDocumentRelatedToSearchTerm,
        - [<entire contents of the Internet]))

Difference between map and xmap?

- The team behind Erlang published results concerning the difference between map and xmap
  - They make a distinction between
    - CPU-bound computations with little message passing vs.
    - lightweight computations with lots of message passing
- With the former, xmap provides linear speed-up (10 CPUs provides a 10x speed-up, then declining) over map
  - the latter less so (10 CPUs provided 4x speed-up)
  - Indeed, xmap's performance in the latter case tends to max out at 4x no matter how many CPUs were added

Agent Model

- The functional language Erlang is credited with creating an approach to concurrency known as the agent model
  - A concurrent program consists of a set of agents
    - Each agent has its own set of data structures that are not shared with other agents
  - Agents can perform computations and send messages
  - Messages sit in an actor’s mailbox until it is ready to process them; they are always processed one at a time
    - An actor does not block when sending a message
    - An actor is not interrupted when a message arrives

Examples

- Examples will be presented in Scala
  - Scala is a language which nicely combines both the imperative and functional programming styles
  - It is implemented on top of Java and thus is cross platform
    - I won’t spend much time explaining Scala; I’ll just focus on the agent model

Example 1

- import scala.actors._
- object SillyActor extends Actor {
  - def act() {
    - for (i <- 1 to 5) {
      - println(“I’m acting!”)
      - Thread.sleep(1000)
      - }
    - }
  - }
- }

- object SeriousActor extends Actor {
  - def act() {
    - for (i <- 1 to 5) {
      - println(“To be or not to be”)
      - Thread.sleep(1000)
    - }
  - }
- }

Running Example 1

- SillyActor.start() ; SeriousActor.start()
- Demo
  - From this example we can see that Actor is a class that can be sub-classed (just like Thread in Java)
  - You start an actor by calling start()
  - At some point, the scheduler calls the actor's act() method
    - The actor will be active until that method returns
    - This is just like Thread's run() method, only the name has changed

Processing Messages

- To process a message, an actor must use either the receive or react keyword
  - react is a special case of receive that we'll discuss below
- You can think of receive as a "switch" statement that specifies the structure of the different type of messages it wants to receive
  - When an actor calls receive, it looks at the mailbox and attempts to find a waiting message that matches one of the branches of the "switch" statement
    - it processes the first match that it finds

Example

- val echoActor = actor {
  - while (true) {
    - receive {
      - case msg =>
        - println(“received message: “ + msg)
    - }
  - }
- }
- This actor loops forever and prints out any message it receives

A message is sent with the ! operator:

echoActor ! “hi there”
echoActor ! 25

Demo

Conserve Threads

- When an act() method uses the receive keyword, it tells the scala run-time system that this actor needs its own thread
  - The actor may be spending its time switching between processing messages and performing a long computation
- Since threads in Java are not cheap, scala provides the react keyword to tell the runtime that all this thread does is react to messages
  - This means it spends most of its time blocked
    - Scala uses this information to assign “react actors” to a single thread, thus conserving threads in the overall system

Example

- object NameResolver extends Actor {
  - …
  - def act() {
    - react {
      - case (name: String, actor: Actor) =>
        - actor ! getIp(name)
        - act()
      - case “EXIT” =>
        - println(“quitting”)
    - }
    - …

Note: no explicit loop; that’s because react doesn’t return (enables sharing of multiple actors on a single thread)

instead, react must call act() if it wants to keep waiting for messages

Results

- To test Scala's claim that react helps conserve threads
  - I wrote a program that can create a specified number of NameResolvers that either
    - use receive or
    - use react
- Results: when creating 100 NameResolvers
  - using receive: 104 threads created
  - using react: 7 threads created (!)

Returning to the Ornamental Garden

- With the Agent model of concurrency, you can easily avoid interference problems
  - Here's an example of the ornamental garden problem
    - No need for mutual exclusion: create two agents that act as turnstiles and have them send increment messages to a shared counter agent
- We saw this last week in lecture 18

Hiding Concurrency…

- An agent-based approach can start to hide concurrency from the developer
  - the implicit threading approach…
- as we will see in this next example
  - TopStock -> retrieve stock quotes from a specified set of stocks for a specified year and lists the one with the highest 52-week price.
  - The quotes are requested in parallel and handled when the main thread is ready for them
- Demonstration

Summary: Alternative Approach

- We have looked at a few alternative models to the “locks and shared data” model of concurrency that
  - draw on functional programming techniques
  - do not allow threads to share data
  - allow threads to communicate via asynchronous messages
- Deadlock and Race conditions are still possible in this model but harder to achieve
  - However, interference is simply not possible in this model
- Functional techniques seem like a promising method for tackling concurrency on multi-core hardware

Wrapping Up

- Threading Libraries
  - Implicit threading libraries manage threads for you
  - Explicit threading libraries provide primitives, the rest is up to you
- Introduced the Scala Agent Model
  - which is an example of an implicit threading library
    - Using receive() in an agent typically causes the creation of a thread
    - Using react() in an agent typically causes the agent to be share a thread with other “reactive” agents

Coming Up

- Lecture 20: Testing and Continuous Integration
  - Read Chapter 7 of the Head First Software Development textbook