Concurrency in Programming Languages

CSC 458 Presentation
Piotr Faliszewski
pfali@cs.rochester.edu

Presentation Plan

- Theoretical models of concurrent computation
  - Nondeterministic Turing Machine
  - Alternating Turing Machine
- Imperative and functional programming languages
- Case study:
  - Concurrent ML
  - OpenMP (Fortran/C/C++)
  - Erlang
  - Split-C
  - MultiLisp
  - Java

Functional versus Imperative Languages

- From comp.lang.functional:
  - `Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.`
  - On the other hand, imperatve languages specify ways of forming simple commands for the computer to execute in a given order.

Functional versus Imperative

- Haskell

```
sum [1..10]
```

- Scheme/Lisp

```
(define sum
   (lambda (from total)
      (if (= 0 from)
         total
         (sum (- from 1) (+ total from)))))
(sum 10 0)
```

- C/C++

```
total = 0;
for (i=1; i<=10; ++i)
   total += i;
```

Case Study of Programming Languages

- Interesting languages:
  - Concurrent ML – concurrency added to a well-established functional language
  - Erlang – industry's response to the need of a practical programming language for concurrent applications
  - MultiLisp – semantics for Lisp that facilitate parallel programming
  - OpenMP – high performance computing community's way of expressing parallelism in C/C++ and Fortran
  - Split-C – dialect of C that facilitates programming for distributed memory multiprocessor
  - Java – general purpose programming language

Concurrent ML – Introduction

- Standard ML
  - Safe, modular, strict, functional, polymorphic programming language
    - Safe – no core dumps
    - Modular – mechanism for modularization of programs
    - Strict – call by value
    - Functional – has higher order functions
    - Polymorphic – generic functions/data types (e.g., the same function can compute the length of a list of strings and the length of a list of ints etc.)
  - Compile-time type checking
  - Garbage collection
  - If needed, can pretend that is not as functional as it seems at first
    - C++ of functional languages

Standard ML – History

- Originally developed in the early 70s in Edinbourgh
- The name, ML, comes from Meta-Language
- Combines features of Algol and Lisp-like languages
- Its first goal was to facilitate the development of a theorem proving package

Standard ML – sequential code

- Simple function:

```
- val add2 = fn (x) => x+2;
val add2 = fn : int -> int
- add2(4);
val it = 6 : int
```

- Fibonacci Sequence (recursive function):

```
- val rec fib = fn 0 => 1 |
=   1 => 1 |
=   N => fib(N-1) + fib(N-2);
val fib = fn : int -> int
- fib 5;
val it = 8 : int
```

Standard ML – Higher Order Functions

- Applying a function to all elements of a list

```
- val map = fn (f, nil) => nil |
=   (f, h::t) => (f h)::(map (f, t));
val map = fn : ('a -> 'b) * 'a list -> 'b list
- map ( add2, [1,2,3] );
val it = [3,4,5] : int list
```

- We could also just return a function that would perform the operation...
  - ...but we are to talk about concurrency, not ML ☺

Concurrent ML

- Concurrency in ML
  - Achieved mostly by a library
  - CML programs spawn processes when they need them
  - Various synchronization mechanisms:
    - Communication channels
    - Events
    - Mailboxes
    - SyncVars

Concurrent ML

- Communication Channels
  - `send/recv` functions (blocking)
  - `sendPoll/recvPoll` functions (nonblocking)
  - `sendEvt/recvEvt` – create events associated with sending/receiving messages
  - Multicast channels also available
- Events
  - `sync` – synchronize on an event
  - ... and many other event related functions that could appear in any library supporting concurrency

Concurrent ML

- SyncVars
  - Variables that can be either filled or empty;
    - Synchronization on reading from an empty variable
  - Two flavors
    - Regular – a value can be put in only once
    - Reusable – a value can be removed, and a new one can be put in (a locker ☺)
  - Blocking and nonblocking operations available
  - Events for typical operations available.

Concurrent ML

- MailBoxes
  - A marriage of SyncVars and channels
    - Producer may put packages into the mailbox
    - Consumer may pick them up
    - No limit on the amount of packages in the mailbox, so if produces is much faster then we run out of memory
  - Operations possible on a MailBox:

    send – always nonblocking

    recv/recvPoll – blocking/nonblocking receive
  - Events associated with mailboxes (receiving)

Concurrent ML

- Summary
  - Nice functional programming language...
  - ... but the syntax has no support for concurrency
  - Concurrency was added using a well designed library, but it did not facilitate concurrent programming too much (no more than pthreads facilitated concurrent programming in C/C++).
  - Functional properties of ML seem to be leaving a lot of space for advanced support of parallel programming templates, but CML does not have them (at least not in a out-of-the-box fashion)

Erlang

- Erlang – developed by Ericsson to facilitate the development of their telecommunication systems.
- Designed as a functional, concurrent programming language
  - Designed by a company – a very pragmatic approach was taken
  - Language design went together with language implementation
    - Features that Ericsson programmers did not use were being removed from the language;
    - Features solving common problems were being added.
- First implementation had only a Prolog based interpreter.
  - Currently, compilers are available.

Erlang – Sequential Code

- Erlang Tutorial: Simple nonrecursive function (and typical layout of a source file)

```
-module(dbl).
-export([double/1]).

double(X) ->
    2 * X.
```

Erlang – Sequential Code

- Recursive Function:

```
-module(tut).
-export([fib/1]).

fib(1) ->
    1;
fib(2) ->
    2;
fib(N) ->
    fib(N-2)+ fib(N - 1).
```

Erlang – Sequential Code

- Erlang Tutorial: Length conversion function:

```
-module(tut3).
-export([convert_length/1]).

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

Erlang – Concurrency

- Erlang can spawn many processed
  - No support for shared memory
  - All communication through messages
  - Messages may be addressed to specific PIDs...
  - ...or sent to special named processes
- Remember! This language was developed by a telecommunication company to satisfy its own needs ☺

Erlang – Concurrency

- Erlang Tutorial: Spawning a process:

```
-module(tut14).
-export([start/0, say_something/2]).

say_something(What, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

Erlang – Concurrency

- Erlang Tutorial: Result:

```
1> tut14:start().
hello
goodbye
<0.63.0>
hello
goodbye
hello
goodbye
```

Erlang – Message Passing

- Simple message passing
  - Ping sends a series of messages to Pong
  - Pong sends them back
- Program will run on two different nodes
- We will handle the case when no reply arrives in sensible time

Erlang – Message Passing (Erlang Tutorial)

```
ping(0, Pong_Node) ->
 io:format("ping finished~n", []);
ping(N, Pong_Node) ->
 {pong, Pong_Node} ! {ping, self()},
 receive
   pong ->
     io:format("Got Pong~n",[])
 end,
 ping(N - 1, Pong_Node).
```

```
start_pong() ->
  register(pong, spawn(tut19, pong, [])).
start_ping(Pong_Node) ->
  spawn(tut19, ping, [3, Pong_Node]).
```

```
pong() ->
  receive
    {ping, Ping_PID} ->
      io:format(„Got ping~n", []),
    Ping_PID ! pong,
    pong()
  after 5000 ->
    io:format("Timed out~n", [])
  end.
```

MultiLisp

- Extension to Lisp developed at MIT for their parallel computer
  - `(pcall A B C)`
    - Evaluates A, B, and C in parallel
  - Futures
    - `(future X)`
    - Returns immediately, and returns a value „future"
    - When X evaluates, the actual result is substituted for the „future"
    - If the value is needed earler then the program waits
- No presentation about functional programming languages is complete without a reference to Lisp ☺

OpenMP

- Not a language as such
  - A set of compiler directives to express parallelism
- Support for:
  - Fortran (77, 90, 95)
  - C/C++
- Goals:
  - Simplify development of shared-memory parallel programs
  - Facilitate parallelization of old sequential programs

OpenMP – History

- Release History
  - October 1997: Fortran version 1.0
  - Late 1998: C/C++ version 1.0
  - June 2000: Fortran version 2.0
  - April 2002: C/C++ version 2.0
  - ...currently version 2.5 is being prepared

OpenMP – Ideas

- Fork-Join programming model
  - Driven by compiler directives
  - ... and some helper functions
  - ... and environment variables
- Parallel sections executed by a team of threads
  - Threads numbered from 0 to N-1.
  - Master thread always has number 0
  - Master Thread is the only one to continue after the parallel section
  - Nested parallelism (not necessarily implemented)

OpenMP – Ideas

- Very easy to use. Most important directives:
  - `parallel` – start a parallel section
  - `for` – distribute loop iterations between threads (data parallelism)
  - `sections` – distribute fragments of code between threads (functional/task parallelism)
  - `critical` – critical section
  - `master/single` – section executed only by the master thread/one thread

OpenMP

- Simple OpenMP program (OpenMP Tutorial)

```
#include <omp.h>
main ()  {
int nthreads, tid;
#pragma omp parallel private(tid)
  {
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid);
   if (tid == 0) {
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
     }
  }
}
```

OpenMP

- Parallel loop (OpenMP Tutorial)

```
#pragma omp parallel shared(a,b,c,chunk) \
  private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }
}
```

OpenMP

- Parallel loop, but shorter (OpenMP Tutorial)

```
#pragma omp parallel for \
    shared(a,b,c,chunk) private(i) \
    schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
```

OpenMP

- Functional/Task parallelism

```
#pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N/2; i++)
      c[i] = a[i] + b[i];
    #pragma omp section
    for (i=N/2; i < N; i++)
      c[i] = a[i] + b[i];
    }
```

OpenMP – Synchronization

- Implicit barrier at the end of most of the constructs.
- Some explicit constructs exist, though.

```
#pragma omp parallel shared(x)
  {
  ...
  #pragma omp critical
  x = x + 1;
  ...
  }
```

```
#pragma omp parallel shared(x)
{
  ...
  if(x == 0)
  {
   #pragma omp barrier
  }
  ...
}
```

OpenMP

- ... and many other cool features:
  - Reduce operation
  - Local/shared variables
    - How to initialise thread-local variables
    - How to restore values in the master thread variables after all threads finish a parallel section
    - etc.

Split-C

- Parallel C for distributed memory machines
- Access to underlying machine "with no surprises„
- All programs are parallel from the start to the end
  - There are always `PROCS` threads.
  - Every thread has a `MYPROC` ranging from `0` to `PROCS-1`
- Small set of synchronization primitives.
- Two dimensional address space
  - Local memory
  - Global memory
    - `global` pointers

Split-C

- New language features:
  - Global pointers
  - Spread pointers (for pointer arithmetic on global pointers)
  - Signalling assignment (store)
    - `all_store_sync`
    - `store_sync`
  - Split-phase assignment
    - `sync`
  - Bulk assignment
  - Atomic operations

Split-C

- Sample program computing P

```
splitc_main( ){
   int trials, mytrials,i,hits;
   int total;
   double pi;
  trials = ...  // obtain global number of trials
   mytrials = trials/PROCS + 1;
   for( int i=0; i<mytrials; i++)
       hits += hit();
   total = all_reduce_to_one_add(hits);
   on_one {
       pi = 4.0*total/(mytrial*PROCS);
   }
}
```

Split-C

- Summary
  - A little similar to OpenMP for C
  - ... but for distributed memory machines
  - Not so much fun to program with

Java

- We all know what Java is...
- Supports synchronization via monitors
  - Every object is a monitor
  - Every `synchronized` method becomes owner of the monitor upon entry, and releases it upon return
    - But there are also functions to manage who the monitor
- Fun Fact: Java does not even guarantee that threads make forward progress
  - But, we can hope that JavaVM tries to be fair, and respects thread priorities

Java

- Communication between threads
  - Shared memory
  - Interruptions
- Synchronization
  - Monitors
    - Synchronized methods (lock on yourself)
      - `synchronized` is not a part of method's signature
    - Synchronized blocks (lock on a given object)
    - Locking static fields by locking on the `Class` object
  - Join operation

Java

- Monitor functions:
  - `wait` – thread holding the monitor releases it and waits to be notified;
  - `notify` – wakes up one waiting thread, but does not give him the monitor; it has to compete for it as everyone else;
  - `notifyAll` – as above, but for all waiting threads

Java 1.5

- New cool libraries in Java 1.5
  - `java.util.concurrent`
    - Thread pools
    - Concurrent collections
    - Timing
    - Synchronization classes
      - Semaphores
      - Barriers
      - Futures (representation of the results of asynchronous computation)
      - Etc.

Summary

- Different languages, different applications
  - Erlang – satisfies industry's need of a language for quick development of distributed systems
  - OpenMP – high performance computing on shared memory multiprocessors
  - Split-C – high performance computing on distributed memory multiprocessors
  - Java – general purpose programming language with native support for parallel computation