Lecture 2: Power of Concurrency

• In this lecture, we address two major questions?
  − Can we do things with concurrency (and parallel platforms) that we could not do with conventional serial programs?
  − What is the basic difficulty associated with writing concurrent programs?
Execution Platforms

- Concurrent programs execute on:
  - Conventional Serial Platforms: Key motivation here is service time, fairness, preventing livelocks.
  - Parallel Hardware: Multicore processors, coprocessors, symmetric multiprocessors, clusters: Key motivation here is speed.
  - Distributed Platforms: Services, P2P environments: Key motivation here is the need to integrate physically distributed resources.
Digression: Parallel Hardware

- Needed?
Power of Concurrency

- Can a concurrent program/hardware do things that a serial program/hardware cannot do?
  - We answer this formally by demonstrating the equivalence between a single-tape and n-tape Turing machine.
Turing Machines

- Serve as the basis for evaluating the “computability” of a problem.
- Turing machines provide a formal model that can compute everything a traditional computer can, and vice versa.
- We can use this formal model to reason about the computability of a given problem.
Turing Machines

• Example: Devise an algorithm to determine if an arbitrary polynomial has integral roots.

• Can show that there cannot be an algorithm for the above problem!
Turing Machines

- A machine consisting of a tape of infinite length and a read-write head which can move left and right across the tape.
Turing Machines

• Tape initially contains an input string followed by blanks
• When started, a Turing Machine executes a series of discrete transitions, as determined by its transition function and by the initial characters on the tape
Turing Machines

- Transitions depend on current state and character under tape head.
- Transitions lead to a new state, write a new character on the tape, and move the tape head one position left or right.
- The machine stops on a transition to a special HALT state (accept or reject).
- The machine may never reach this state.
Turing Machines

- This TM when reading an **a** will move **right** one square stays in the same start state. When it scans a **b**, it will change this symbol to an **a** and go into the other state (accept state).

  $a \rightarrow a, R$
  
  $b \rightarrow a, R$

  $a \rightarrow a, R$ means TM reads the symbol **a**, it replaces it with **a** and moves head to **right**
Turing Machines

• A TM that tests for memberships in the language
  - \( A = \{w\#w \mid w \text{ belongs to } \{0,1\}^*\} \)
  - A language that has two identical strings (w) separated by a hash mark.

• Idea:
  - Zig-zag across tape, crossing off matching symbols
Turing Machines

- Tape head starts over leftmost symbol
- Record symbol in control and overwrite X
- Scan right: reject if blank encountered before #
- When # encountered, move right one space
- If symbols don’t match, reject
Turing Machines

- Overwrite X
- Scan left, past # to X
- Move one space right
- Record symbol and overwrite X
- When # encountered, move right one space
- If symbols don’t match, reject
Turing Machines

- Finally scan left
- If a or b encountered, reject
- When blank encountered, accept
Turing Machines: Formal Definition

- A Turing Machine is a 7-tuple \((Q, \Sigma, T, \xi, q_0, q_{\text{accept}}, q_{\text{reject}})\), where
  - \(Q\) is a finite set of states
  - \(\Sigma\) is a finite set of symbols called the alphabet
  - \(T\) is the tape alphabet, where \(\_\) belongs to \(T\), and \(\Sigma \subseteq T\)
  - \(\xi : Q \times T \rightarrow Q \times T \times \{L, R\}\) is the transition function
  - \(q_0 \in Q\) is the start state
  - \(q_{\text{accept}} \subseteq Q\) is the accept state
  - \(q_{\text{reject}} \subseteq Q\) is the reject state, where \(q_{\text{accept}} \neq q_{\text{reject}}\)
Turing Machines: Formal Definition

\[ \xi : Q \times T \rightarrow Q \times T \times \{L,R\} \text{ is the transition function} \]
\[ \xi(q,a) = (r,b,L) \text{ means} \]

"in state \( q \) where head reads tape symbol \( a \), the machine overwrites \( a \) with \( b \), enters state \( r \), and moves the head left"
Turing Machines: Formal Definition

- \( M = (Q, \Sigma, T, \xi, q_0, q_{\text{accept}}, q_{\text{reject}}) \) computes as follows

  - Input \( w = w_1w_2 \ldots w_n \) is on leftmost \( n \) tape squares
  - Rest of tape is blank –
  - Head is on leftmost square of tape
Turing Machines: Formal Definition

- \( M = (Q, \Sigma, T, \xi, q_0, q_{\text{accept}}, q_{\text{accept}}) \)
- When computation starts
  - M Proceeds according to transition function \( \xi \)
  - If M tries to move head beyond left-hand-end of tape, it doesn’t move
  - Computation continues until \( q_{\text{accept}} \) or \( q_{\text{accept}} \) is reached
  - Otherwise M runs forever
Turing Machines: Formal Definition

- $uarbv$ yields $upacv$ if
  - $\xi(r,b) = (p,c,L)$

- $uarbv$ yields $uacpv$ if
  - $\xi(r,b) = (p,c,R)$

- Special cases: $rbv$ yields $pcv$ if
  - $\xi(r,b) = (p,c,L)$

- $wr$ is the same as $wr--$
Turing Machines: Formal Definition

- TM $M$ accepts input $w$ if a sequence of configurations $C_1, C_2, \ldots, C_k$ exist
  - $C_1$ is start configuration of $M$ on $w$
  - Each $C_i$ yields $C_{i+1}$
  - $C_k$ is an accepting configuration
- The collection of strings that $M$ accepts is the *language* of $M$, denoted by $L(M)$
Turing Machines: Formal Definition

- Definition: A language is *recursively enumerable* if some TM accepts it.
- On an input to a TM we may
  - accept
  - reject
  - loop (run for ever)
- Not very practical: never know if TM will halt
Turing Machines: Formal Definition

- Definition: A TM *decides* a language if it always halts in an accept or reject state. Such a TM is called a *decider*.

- Definition: A language is *decidable* if some TM decides it.
  - Some textbooks use *recursive* instead of *decidable*.

- Therefore, every decidable language is enumerable, but not the reverse!
Turing Machines: Formal Definition

− Here is a decidable language
  • $L = \{a^ib^jc^k \mid ix = j, l,j,k > 0\}$
− Because there exist a TM that decides it
− How?
Turing Machines: Formal Definition

- What is not decidable?
- \( D = \{ p \mid p \text{ is a polynomial with an integral root} \} \)
  - Determine whether the set \( D \) is decidable
  - (Can’t do it)
- We can show that \( D \) is Turing-recognizable
Consider Hilbert’s problem only for variable X

- $D_1 = \{p \mid p$ is a polynomial over $x$ with integral roots$\}$
- $TM(D_1) :$ Evaluate $p$ with $x$ set successively to the values $0, 1, -1, 2, -3, 3, -3, \ldots$ if at any point the polynomial evaluates to $0$, accept
- If $D_1$ has an integral root, this TM will eventually find it
- If $D_1$ has no integral root, this TM will run forever
- This TM is a recognizer not a decider
Undecidable Problems

- Given a C program (or a program in any programming language, really) that prints "hello, world" is there another program that can test if a program given as input prints "hello, world"?
Undecidable Problems

- This is tougher than it may sound at first glance. For some programs it is easy to determine if it prints hello world. Here is perhaps the simplest:

```c
#include "stdio.h"
void main()
{
    printf("hello, world\n");
}
```
Undecidable Problems

- It would be fairly easy to write a program to test to see if another program consisting solely of printf statements will output “hello, world”. But what we want is a program that can take any arbitrary program and determine if it prints “hello, world”.

- This is much more difficult.
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1,_,main(-86,0,a+1)+a)):
1,t<3?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?<_13?
main(2,_,+1,"%s %d %d\n"):<16:t<72?main(_,t,
"@n'+,#'"{w+w#cdnr/+,{r/*de}+,/*+,/w{%+,.w#q#n+,./#l,+,/n{n+n,;++n+,/#
};q#n+,/+k#;+,."d'3,}{w+K w'K:+)e#;dq#'l \nq#+d'K#/+k#:q#r)eKK{nI}'#:q#n'}#w')}{nI}' +#d)w' i;#
}{nI}/n{n#: r(#w'r nc{nI}'#{l,+K {rw' iK{[{nI}'#w#q#n'wk nw' 
}|w{KK{nI}/w{%l##w' i;}:{nI}'{q#'ld;r#n'}{nlwbl/*de}'c 
;;{nI}'rw]'/+,.,##'*}#nc',#nw]'/+kd'+e;}#'rdq#w! nr'/ ') }+{rI'#{n' '})# 
}+}##(!"/"
:t<50?___=*a?putchar(31[a]):main(-65,_,a+1):main("a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):a="'/"llmain(0,main(-61,"a,
"!ek;dc i@bK'(q)=[w]*n+r3#l,{}:\nuwloca-O;m .vptks.fxntdCeghiry"},a+1);
}
Undecidable Problems

pinky.cs.purdue.edu 326 % ./a.out
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

.......
Undecidable Problems

- Problem: Create a program that determines if any arbitrary program prints “hello world”
- We can show there is no program to solve that problem (i.e. it is undecidable)

- Suppose that there were such a program H, the “hello-world-tester.”
- H takes as input a program P and an input file I for that program, and tells whether P, with input I, prints “hello world” and outputs “yes” if it does, “no” if it does not
Undecidable Problems

H
Hello-world tester

I
P
yes
no
Undecidable Problems

- Next we modify H to a new program H1 that acts like H, but when H prints no, H1 prints “hello, world.”.
- To do this, we need to find where “no” is printed and instead output “hello world” instead:

```
  I
  P
  H1
    Hello-world tester
```

- yes
- hello, world
Undecidable Problems

- Next modify H1 to H2. The program H2 takes only one input, P2, instead of both P and I.
- To do this, the new input P2 must include the data input I and the program P.
- The program P and data input I are all stored in a buffer in program H2. H2 then simulates H1, but whenever H1 reads input, H2 feeds the input from the buffered copy. H2 can maintain two index pointers into the buffered data to know what current data and code should be read next:
Undecidable Problems

- However, H2 cannot exist. If it did, what would H2(H2) do?
- That is, we give H2 as input to itself...

If H2 on the left outputs = “yes”, then H2 given H2 as input will print “hello, world”. But we just supposed that the first output H2 makes is “yes” and not “hello world”.

The situation is paradoxical and we conclude that H2 cannot exist and this problem is undecidable.
Multitape Turing Machines

- A multitape Turing machine is like an ordinary TM but it has several tapes instead of one tape.
- Initially the input starts on tape 1 and the other tapes are blank.
- The transition function is changed to allow for reading, writing, and moving the heads on all the tapes simultaneously.
  - This means we could read on multiples tape and move in different directions on each tape as well as write a different symbol on each tape, all in one move.
Theorem: A multitape TM is equivalent in power to an ordinary TM. Recall that two TM’s are equivalent if they recognize the same language. We can show how to convert a multitape TM, M, to a single tape TM, S:
Multitape Turing Machines

- Say that M has k tapes.
  - Create the TM S to simulate having k tapes by interleaving the information on each of the k tapes on its single tape
  - Use a new symbol # as a delimiter to separate the contents of each tape
  - S must also keep track of the location on each of the simulated heads
Non-Deterministic Turing Machines

- Theorem: Can be shown that these are identical to single tape turing machines as well.