Deterministic Concurrency

Candidacy Exam

Nalini Vasudevan

Columbia University
Motivation
Why Parallelism?

Past Vs. Future

- Power wall: Power vs. transistors
- Static vs Dynamic power
- Memory wall: Multiply vs. load and stores
- Faster sequential computer?
- Clock frequency

[Asanovic et al., The Landscape of Parallel Computing Research, 2006]
Programming Models

- Performance
- Psychological: Ease of use
- No. of processors
- Types of Parallelism

---

[Background and Jargon of Parallel Computing, Patterns for Parallel Programming, 2004]

---

Task level

- P1: `main()`
  - P2: `foo()`
  - P3: `bar()`

Data level

- P1: `a[0]`
  - P2: `a[1]`
  - P3: `a[2]`
Programming Models

Nested Data Level Parallelism

function quicksort(a) =
    if (#a < 2) then a
    else
        let pivot = a[#a/2];
        lesser = {e in a| e < pivot};
        equal = {e in a| e == pivot};
        greater = {e in a| e > pivot};
        result = {quicksort(v): v in [lesser,greater]};
        in result[0] ++ equal ++ result[1];

quicksort([8, 14, -8, -9, 5, -9, -3, 0, 17, 19]);

[Programming Parallel Algorithms, Blelloch, ACM Communications, 1996.]
The Data Race Problem

```c
int x = 1;
bar() {
  x++;
}
foo() {
  spawn(bar);
  x++;
}
```

ld x
add x, x, 1
st x
The Data Race Problem

```
int x = 1;
bar() {
    x++;  
}
foo() {
    spawn(bar);
    x++;  
}
```

```
int x = 1;
bar() {
    lock(x);
    x++;  
    unlock(x);
}
foo() {
    spawn(bar);
    lock(x);
    x++;  
    unlock(x);
}
```

ld x
add x, x, 1
st x
High Level Data Races

\[\begin{align*}
\text{T1} & : & \text{lock}(m) \\
& & x = x \times 2; \\
& & y = x + 2; \\
& & \text{unlock}(m)
\end{align*}\]

\[\begin{align*}
\text{T2} & : & \text{lock}(m) \\
& & \text{read}(x); \\
& & \text{unlock}(m) \\
& & \text{..} \\
& & \text{lock}(m) \\
& & \text{read}(y); \\
& & \text{unlock}(m)
\end{align*}\]

[Artho et al., High Level Data Races, 2003]
Non-determinism

```c
int x = 1;
void bar() {
    lock(m);
    x = x*2;
    unlock(m);
}
void foo() {
    spawn(bar);
    lock(m);
    x++;  
    unlock(m);
    x = ?;
}
```

Output: 2, 3 or 4

```
x = 1
bar: x = 1 * 2 = 2
foo: x = 2
foo: x = 3
bar: x = 2*2 = 4
```
Concurrent Programming Models
The Cilk Programming Model

- Each thread is non-blocking
- Downward edges denote spawned threads
- Horizontal edges denote successors
- Work stealing scheduler

- Non-deterministic: Allows access to global resources

[Blumofe et al., Cilk: An Efficient Multithreaded Runtime System, 1995]
An Example in Cilk

```cilk
thread sum() {
   cont int x, y;
   spawn_next add(?x, ?y);
   spawn getx (x);
   spawn gety (y);
}

thread getx (cont int x) {
   send argument (x, 5);
}

thread gety (cont int y) {
   send argument (y, 3);
}

thread add(int x, int y) {
   printf("%d", x + y);
}
```

![Diagram showing the flow of execution in Cilk](image)
The X10 Programming Language

- Concurrent programming model
- Activities are light weight threads
- Places represent distributed memory

[Charles et al., X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, 2005]
The X10 Programming Language

- Activities created using `async`

  ```x10
  async {
  /* Body of async executed locally */
  }
  ```

  ```x10
  async (p2) {
  /* Body of async executed at p2 */
  }
  ```

- Synchronization between activities through
  - `finish`
  - `atomic`
  - `clocks`

- Improper synchronization can lead to races
Deterministic Concurrent Programming Models
Kahn Networks

```c
void f(int in u, int in v) {
    int i;
    bool b = true;
    while (1) {
        i = if (b) then wait (u) else wait (v);
        print (i);
        b = ~b;
    }
}

void g(int out u) {
    u = 1;
    while (1) {
        send u;
    }
}

void h(int out v) {
    v = 0;
    while (1) {
        send v;
    }
}

// Body of main program;
    f(u, v) par g(u) par h(v);
```
Properties of Kahn Networks

- **Wait**: Waits for the sender to send data
- **Send**: Nothing prevents a process to send
  - No two stations are allowed to send data on the same channel
- Behaves like FIFO queues
- Deterministic behavior

Problem: Unbounded Buffers
Synchronous Data Flow

- Subset of Kahn Networks
- Actor: fires by removing tokens from its input edges and producing tokens on its output edges
- Edges represent communication channels, implemented as FIFO
- Each actor produces and consumes a fixed number (known apriori)
- Model suitable for DSP applications
The StreamIt Model

• A synchronous data flow language
• A structured model for streams

```
pipeline FMRadio {
    add DataSource();
    add LowPassFilter();
    add FMDemodulator();
    add Equalizer();
    add Speaker();
}
```
The StreamIt Model

- Filter
  - Autonomous unit of computation
  - No access to global resources
  - Communicates through FIFO channels: pop(), peek(index), push(value)
Esterel: A synchronous model

- Suited for reactive systems
- Follows the synchrony hypothesis
   Let $t$ be the communication time
   - $t$ arbitrary $\rightarrow$ asynchrony
   - $t$ predictable $\rightarrow$ vibration
   - $t = 0$ $\rightarrow$ synchrony
- Notion of a global clock

---

[Berry et al., The ESTEREL Synchronous Programming Language, 1992]
An Example in Esterel

input COIN, TEA_BUTTON, COFFEE_BUTTON;
output SERVE_TEA, SERVE_COFFEE;
loop
    await COIN;
    await
    await
        case TEA_BUTTON do emit SERVE_TEA;
        case COFFEE_BUTTON do emit SERVE_COFFEE;
    end await;
end loop;
Compiling Esterel

\[
\text{loop} \\
\text{emit A;} \\
\text{await C;} \\
\text{emit B;} \\
\text{pause} \\
\text{end}
\]

[Edwards, Tutorial: Compiling Concurrent Languages for Sequential Processors, TODAES, 2003]
Compiling Esterel

```
void tick() {
    static int s = 0;
    A = B = 0;
    switch (s) {
        case 0:
            A = 1;
            s = 1;
            break;
        case 1:
            if (C) {
                B = 1;
                s = 0;
            }
            break;
    }
}

main() {
    for(;;)
        tick();
}
```

[Edwards, Tutorial: Compiling Concurrent Languages for Sequential Processors, TODAES, 2003]
void tick() {
    static int s = 0;
    A = B = 0;
    switch (s) {
        case 0:
            A = 1;
            s = 1;
            break;
        case 1:
            if (C) {
                B = 1;
                s = 0;
            }
            break;
    }
}

main() {
    for(;;) tick();
}
The SHIM Programming Language
The SHIM Model

- Stands for *Software Hardware Integration Medium*
- Race free, scheduling independent, concurrent model
- Blocking CSP-style rendezvous communication

[Edwards et al., SHIM: A Deterministic Model for Embedded Systems, 2005.]
CSP

- Stands for Communicating Sequential Processes
- Model interactions between processes
- Supports synchronization, concurrency etc.
- Rendezvous, message passing

SHIM = Kahn + CSP
An Example in SHIM

- Blocking: wait for all processes connected to \( c \)

```c
void f(chan int a) { // a is a copy of c
    a = 3; // change local copy
    recv a; // receive (wait for g)
    // a now 5
}
void g(chan int &b) { // b is an alias of c
    b = 5; // sets c
    send b; // send (wait for f)
    // b now 5
}
void main() {
    chan int c = 0;
    f(c); par g(c);
    c = c * 2;
}
```

--

[Tardieu et al., Scheduling-Independent Threads and Exceptions in SHIM]
The Problem

```c
void main() {
    chan int a, b;
    {
        // Task 1
        a = 15, b = 10;
        send a;
        send b;
    } par {
        // Task 2
        int c;
        recv b;
        recv a;
        c = a + b;
    }
}
```
Analysis of Concurrent Programs
Why Analyze?

- Data Race Detection
- Deadlock Detection
- Optimization

Analysis of concurrent programs is generally harder than sequential programs

Abstraction

[Clarke et al., Model checking and Abstraction, 1994]
Model Checking

Abstracted Model  Property to Check

Model Checker

Yes/Dont-know  Error trace
mtype = {NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proc type process(int id) {
    beginning:
    trying:
        state[id] = TRYING;
        if
            :: goto trying;
            :: true;
        fi;
    critical:
        state[id] = CRITICAL;
        if
            :: goto critical;
            :: true;
        fi;
    noncritical:
        state[id] = NONCRITICAL;
        if
            :: goto noncritical;
            :: true;
        fi;
    goto beginning;
}
init { run process(0); run process(1); }

#define cs0 (state[0] == critical )
#define cs1 (state[1] == critical )
!(cs0 && cs1)
Using Petri Nets to analyze concurrent languages

- Reachability analysis (finding deadlocks etc.)
- To generate sequential code from concurrent programs

```c
void ping() {
    for(;;) {
        send a;
        send b;
        send c;
    }
}

void pong() {
    for(;;) {
        send d;
        recv b;
        send e;
    }
}
```

---

[Lin, Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling, DATE 1998
Peterson, Petri Nets, ACM Computing Surveys 9, 1977.]
Using Petri Nets to analyze concurrent languages

- Reachability analysis (finding deadlocks etc.)
- To generate sequential code from concurrent programs

```c
void ping() {
    for(;;) {
        send a;
        send b;
        send c;
    }
}

void pong() {
    for(;;) {
        send d;
        recv b;
        send e;
    }
}
```

---

[Lin, Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling, DATE 1998
Peterson, Petri Nets, ACM Computing Surveys 9, 1977.]
Conclusions

• Problems with concurrent programming languages
  • Concurrent programs are generally harder to analyze
  • Bugs: Data Races, Deadlocks

• Future Work
  • A language that is race-free and deadlock-free
  • An auto-determinizing compiler