Dealing with Concurrency Problems

Nalini Vasudevan
Columbia University
What is the output?

```
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
}
bar(){
    int n;
    n = baz();
    x = x + n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```
What is the output?

```c
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
}
bar(){
    int n;
    n = baz();
    x = x + n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

Data Race
Eliminating Data Races

```c
int x;
foo()
{
  int m;
  m = qux();
  lock(x);
  x = x + m;
  unlock(x);
}
bar()
{
  int n;
  n = baz();
  lock(x);
  x = x + n;
  unlock(x);
}
main()
{
  x = 2;
  foo() par bar();
  print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 3
Eliminating Data Races

```c
int x;
foo()
{
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}
bar()
{
    int n;
    n = baz();
    lock(x);
    x = x + n;
    unlock(x);
}
main()
{
    x = 2;
    foo() par bar();
    print(x);
}
```

```
if m = n = 2
x = (2 + 2) + 2 = 6
```
Another Example

```c
int x;
foo() {
    int m;
    m = qux();
    x = x + m;
}
bar() {
    int n;
    n = baz();
    x = x * n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 4
Eliminating Data Races

```c
int x;
foo(){
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}
bar(){
    int n;
    n = baz();
    lock(x);
    x = x * n;
    unlock(x);
}
main(){
    x = 2;
    foo() par bar();
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 5
Eliminating Data Races

```c
int x;
foo(){
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}
bar(){
    int n;
    n = baz();
    lock(x);
    x = x * n;
    unlock(x);
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 5
Eliminating Data Races

```c
int x;
foo()
{
    int m;
    m = qux();
    lock(x);
    x = x + m;
    unlock(x);
}

bar()
{
    int n;
    n = baz();
    lock(x);
    x = x * n;
    unlock(x);
}

main()
{
    x = 2;
    foo() par bar();
    print(x);
}
```

```
if m = n = 2
x = (2 + 2) * 2 = 8
x = (2 * 2) + 2 = 6
Non-determinism
```
Problem with Locks

```c
int x = 0;
int y = 0;

foo() {
    lockx();
    locky();
    x++;
    y++;
    unlocky();
    unlockx();
}

bar() {
    locky();
    lockx();
    y++;
    x++;
    unlockx();
    unlocky();
}

main() {
    foo(x) par bar(x);
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 6
Problem with Locks

```c
int x = 0;
int y = 0;

foo() {
    lockx();
    locky();
    x++;
    y++;
    unlocky();
    unlockx();
}

bar() {
    locky();
    lockx();
    y++;
    x++;
    unlockx();
    unlocky();
}

main() {
    foo(x) par bar(x);
    print(x);
}
```

Deadlock
Motivation

Parallel Computers
Library Support
Performance
Deadlocks
Parallel Languages
Non-Determinism
Hard-to-Debug
Motivation

Determinism?

Deadlock Freedom?  Efficiency?
Determinism: The SHIM Model

- Stands for *Software Hardware Integration Medium*
- Race free, scheduling independent, concurrent model
- Blocking synchronous rendezvous communication
The SHIM Language

An imperative language with familiar C/Java-like syntax

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
```
Additional Constructs

\[ \text{\textit{stmt}}_1 \, \text{par} \, \text{\textit{stmt}}_2 \quad \text{Run } \text{\textit{stmt}}_1 \, \text{and} \, \text{\textit{stmt}}_2 \, \text{concurrently} \]

\textit{send} \, \textit{var} \quad \text{Send on channel } \textit{var} \\
\textit{recv} \, \textit{var} \quad \text{Receive on channel } \textit{var}
Communication

- 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);
}
```
Compiling to Quad-Core

- Intel Quad Core Machine
- Each task mapped to a pthread
- Example: JPEG decoder

<table>
<thead>
<tr>
<th>Cores</th>
<th>Tasks</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequential</td>
<td>25s</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>16</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9.3</td>
<td>2.7</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>8.7</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8.2</td>
<td>3.05</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8.6</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Run on a 20 MB $21600 \times 10800$ image that expands to 668 MB.
Compiling to Cell [SAC 2009]

- Generated Code for a Heterogeneous Multicore
- Computationally intensive tasks mapped on the SPUs
- Example: FFT

![Bar chart showing execution time vs. number of SPE tasks]

- Observed vs. Ideal execution times for different numbers of SPE tasks.
More Examples in SHIM

```c
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    }
    par {
        // Task q
        int c;
        recv a; // recv a
        recv b; // recv b
        c = a + b;
    }
}
```
The Problem

```c
void main() {
    chan int a, b;
    {
        // Task p
        send a = 5; // send a
        send b = 10; // send b
    } par {
        // Task q
        int c;
        recv b; // recv b
        recv a; // recv a
        c = a + b;
    }
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 16
Static Deadlock Detection

- Scheduling independence
- Data sharing through rendezvous communication
- Asynchronous parts are independent

Just pick one schedule
Deadlocks in SHIM

- Why SHIM? No data races.
- Deadlocks in SHIM are deterministic (always reproducible).
- SHIM’s philosophy: It prefers deadlocks to races.
Deterministic, Deadlock-Free Model

```c
void f(shared int &a) {
    /* a is 1 */
    a = 3;
    /* a is 3, x is still 1 */
    next; /* Apply reduction operator */
    /* a is now 8, x is 8 */
}

void g(shared int &b) {
    /* b is 1 */
    b = 5;
    /* b is 5, x is still 1 */
    next; /* Apply reduction operator */
    /* b is now 8, x is 8 */
}

void h (shared int &c) {
    /* c is 1, x is still 1 */
    next;
    /* c is now 8, x is 8 */
}

main() {
    shared int (+) x = 1;
    /* If there are multiple writers, reduce using the + reduction operator */
    f(x) par g(x) par h(x);
    /* x is 8 */
}
```
Deterministic, Deadlock-free Model

- Histogram Example

```c
void histogram(int a[], int n) {
    int b[10];
    for (int i = 0; i < n; i++) { #par
        int index = a[i];
        b[index]++;
    }
    print (b);
}
```
Deterministic, Deadlock-free Model

- Histogram Example

```c
void histogram(int a[], int n) {
    shared int (+) b[10]
    for (int i = 0; i < n; i++) par {
        int index = a[i];
        b[index] = 1;
        next;
    }
    print (b);
}
```
Deterministic, Deadlock-free Model

Determinism ✓

Deadlock Freedom ✓ Efficiency?
Deterministic, Deadlock-free Model

![Relative Speed Graph]

- AllReduceParallel
- Pipeline
- Convolve
- NQueensPar
- MontyPiParallel
- KMeansScalar
- Histogram
- MergeSort
- Stream
- Prefix
- UTS
- IDEA
- Stencil
- SOR
- Series
- RayTrace
- LUFact
- SparseMatMul

Dealing with Concurrency Problems, Nalini Vasudevan – p. 23
Future Work [PLDI’09 Fun Ideas and Thoughts]

- Parallel Computers
- Library Support
- Parallel Languages
- Performance

A Determinizing Compiler!
The Example

```c
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
}
bar(){
    int n;
    n = baz();
    x = x * n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 25
The Determinizing Compiler’s Role

```c
int x;
foo() {
    int m;
    m = qux();
    x = x + m;
    sync(x);
}
bar() {
    int n;
    n = baz();
    sync(x);
    x = x * n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

Dealing with Concurrency Problems, Nalini Vasudevan – p. 26
The Determinizing Compiler’s Role

```c
int x;
foo(){
    int m;
    m = qux();
    x = x + m;
    sync(x);
}
bar(){
    int n;
    n = baz();
    sync(x);
    x = x * n;
}
main() {
    x = 2;
    foo() par bar();
    print(x);
}
```

if m = n = 2
x = (2 + 2) * 2 = 8
Always!

Dealing with Concurrency Problems, Nalini Vasudevan – p. 26
Scalability

- Synchronization is expensive
- Synchronization = Sequentializing
Scalability

- Synchronization is expensive
- Synchronization = Sequentializing

Suppose a program runs for 100s on a single processor. 80% of the program can be parallelized. What is the speed up on running the program with

1. 2 processors
2. 4 processors
3. 8 processors
Scalability

- Synchronization is expensive
- Synchronization = Sequentializing

Suppose a program runs for 100s on a single processor. 80% of the program can be parallelized. What is the speed up on running the program with

1. 2 processors
2. 4 processors
3. 8 processors

Ans: 1.66, 2.5, 3.33 [Using Amdahl’s law]
The Ultimate Goal

- Determinism ✓
- Deadlock Freedom ✓
- Efficiency ✓