Ensuring Deterministic Concurrency through Compilation
Nalini Vasudevan and Stephen A. Edwards
Columbia University

Parallel Computers
Library Support
Parallel Languages
Performance Races
Deadlocks
Hard to debug

Data Races

```
void f(shared int &a) {
    a = 3;
}

void g(shared int &b) {
    b = 5;
}

main() {
    shared int x = 1;
    spawn f(x);
    spawn g(x);
    print x;
}
```

The above program creates two tasks parallel using the spawn construct. x is being modified concurrently by the two tasks and therefore the program is not race-free.

Our Approach

- Either allow single writes, or allow multiple writes but in a synchronized fashion.
- A write in one phase is available to other tasks only in the next phase.
- Conflicting writes are reduced by an associative, commutative operator.

```
void f(shared int &a) { /* a is f */
    a = 3; /* a becomes 3, x is still 3 */
    next; /* The reduction operator is applied */
    p = a now 8, x is 8 */
}

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

void h(shared int &c) { /* c is 1, x is 3 */
    c = 1; /* c becomes 1, x is 3 */
    next;
    p = c now 8, x is 8 */
}

main() {
    shared int x = 1;
    /* If there are multiple writers, reduce using the + reduction operator */
    spawn f(x);
    spawn g(x);
    spawn h(x);
    sync; /* Wait for f and g to finish */
    print x;
}
```

Example

Histogram Calculation

```
count int N = 100;
count int M = 5;

void list(int value, shared int b[M]) {
    int bucket = value % M;
    b[bucket] = 1;
    next; /* Reduction operator applied here */
}

main() {
    int d[N] = {1};
    shared int x[N];
    for (int i = 0; i < N; i++)
        spawn hist(a[i], b);
    sync; /* Wait for f and g to finish */
}
```

Initial Results

<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>

A JPEG decoder run on a 20 MB 21600×18800 image that expands to 688 MB. (Executed on a Quad-core shared memory machine)

FFT run on a 20 MB audio file. 1024-point FFTs (Executed on a Cell Processor)

Deadlocks

The problem with locks: incorrect usage may lead to deadlocks.

```
void f(shared int &a) {
    lock (p);
    a = 3;
    unlock (p);
}

void g(shared int &b) {
    lock (p);
    b = 5;
    unlock (p);
}

main() {
    shared int x = 1;
    spawn f(x);
    spawn g(x);
    sync; /* Wait for f and g to finish */
    print x;
}
```

We implemented the constructs as a library, and tested it on a systolic filter. (Executed on an Oct-core shared memory machine)

Non-determinism

A remedy to avoid races is to introduce locks.

```
lock p;
void f(shared int &a) {
    lock (p);
    a = 3;
    unlock (p);
}

void g(shared int &b) {
    lock (p);
    b = 5;
    unlock (p);
}

main() {
    shared int x = 1;
    spawn f(x);
    spawn g(x);
    sync; /* Wait for f and g to finish */
    print x;
}
```

Even though x is protected by a lock, the value printed by this program is either 3 or 5 depending on the schedule. Therefore, it is non-deterministic.

By determinism, we mean the output behavior of the program is independent of the scheduling choices (e.g., the operating system) and depends only on the input behavior.