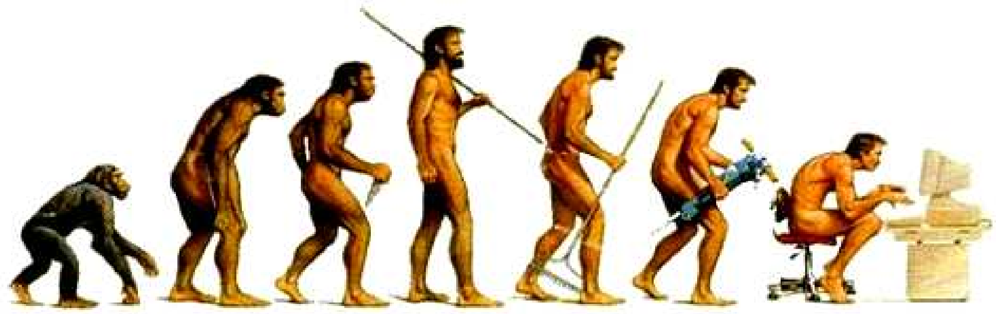
Ensuring Deterministic Concurrency through Compilation

Nalini Vasudevan and Stephen A. Edwards

Columbia University



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);
  sync; /* Wait for f and g to finish */
  print x;
}
```

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

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-determinisitc.

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.

Deadlocks

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

```
lock p, q;

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

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

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

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 1 */
  a = 3; /* a becomes 3 , x is still 1 */
  next; /* The reduction operator is applied */
  /* a is now 8, x is 8 */
}

void g(shared int &b) { /* b is 1 */
  b = 5; /* b becomes 5, x is still 1 */
  next; /* The reduction operator is applied */
 /* 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 */
  spawn f(x);
  spawn g(x);
  spawn h(x);
  sync;
  /* x is 8 */
}
```

The program on the left creates three tasks in parallel f, g and h. f and g are modifying x. Even though f and g are modifying x concurrently, f sees the effect of g only when it executes *next*. Similarly g sees the effect of f only when it executes *next*. When a task executes *next*, it waits for all tasks that share variables with it to also execute *next*. The *next* statement is like a barrier. At this statement, the shared variables are reduced using the reduction operator. In this example, the reduction operator is $+$ because x is declared with a reduction operator $+$. Therefore after the *next* statement, the value of x is 8 and it is reflected everywhere. Function h also rendezvous with f and g by executing *next* and thus it obtains the new value 8.

Example

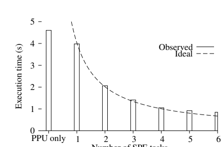
Histogram Calculation

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

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

main() {
  int a[N] = {...};
  shared int (+) b[M];
  for (int i = 0; i < N; i++)
        spawn hist(a[i], b);
  sync; /* Wait for children to finish */
}
```

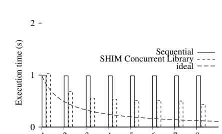
Initial Results

Cores	Tasks	Time	Speedup
1	Sequential	25s	1.0
4	3	16	1.6
4	4	9.3	2.7
4	5	8.7	2.9
4	6	8.2	3.05
4	7	8.6	2.9

A JPEG decoder run on a 20 MB 21600 × 10800 image that expands to 668 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)



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