Mutable state in parallel programs

Matteo Frigo

Cilk Arts

June 16, 2009

The ubiquitous global variable

Fibonacci:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = fib(n-1);
    b = fib(n-2);

    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

The ubiquitous global variable

Ideal world:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = fib(n-1);
    b = fib(n-2);

    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

Real world:

```
int x;

void fib(int n) {
  if (n < 2) {
    x += n;
  } else {
    fib(n-1);
    fib(n-2);
  }
}

int main() {
  x = 0;
  fib(35);
  printf("%d\n", x);
}
```

Parallel Fibonacci

Correct:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = cilk_spawn fib(n-1);
    b = fib(n-2);
    cilk_sync;
    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

Real world:

```
int x;

void fib(int n) {
  if (n < 2) {
    x += n;
  } else {
    fib(n-1);
    fib(n-2);
  }
}

int main() {
  x = 0;
  fib(35);
  printf("%d\n", x);
}
```

Parallel Fibonacci

Correct:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = cilk_spawn fib(n-1);
    b = fib(n-2);
    cilk_sync;
    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

Incorrect:

```
int x;

void fib(int n) {
  if (n < 2) {
    x += n;
  } else {
    cilk_spawn fib(n-1);
    fib(n-2);
  }
}

int main() {
  x = 0;
  fib(35);
  printf("%d\n", x);
}
```

Parallel Fibonacci

Correct:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = cilk_spawn fib(n-1);
    b = fib(n-2);
    cilk_sync;
    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

Incorrect:

```
int x;

void fib(int n) {
  if (n < 2) {
    x += n; // race
  } else {
    cilk_spawn fib(n-1);
    fib(n-2);
  }
}

int main() {
  x = 0;
  fib(35);
  printf("%d\n", x);
}
```

Not so parallel Fibonacci

Fast:

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = cilk_spawn fib(n-1);
    b = fib(n-2);
    cilk_sync;
    return a + b;
  }
}

int main() {
  printf("%d\n", fib(35));
}
```

Slow:

```
int x;

void fib(int n) {
  if (n < 2) {
    atomic_add(&x, n);
  } else {
    cilk_spawn fib(n-1);
    fib(n-2);
  }
}

int main() {
  x = 0;
  fib(35);
  printf("%d\n", x);
}
```

The problem

Want:

- A parallel `fib`.

Subject to:

- You must use the imperative `fib`. You cannot rewrite it.

The problem

Want:

- A parallel `fib`.

Subject to:

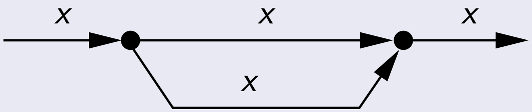
- You must use the imperative `fib`. You cannot rewrite it.

Idea:

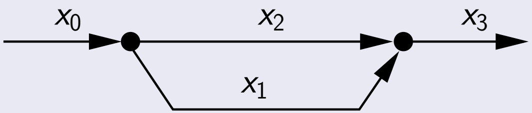
- Redefine the meaning of the state variable `x`.

Hyperobjects

Object: All observers share the same (eq?) view.



Hyperobject: The view depends upon the observer.

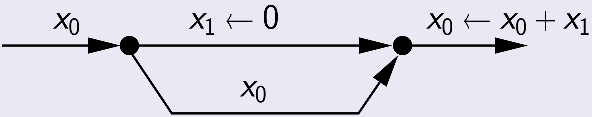


No determinacy races:

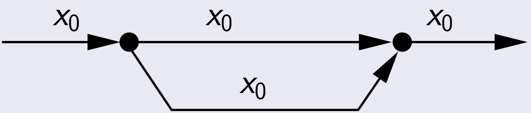
$$A \parallel B \ \Rightarrow\ x_A \neq x_B \qquad \text{(in the eq? sense)}$$

Reducers

Parallel execution:



Serial execution (execute bottommost first):



Nondeterministically pick either execution.

Either way, `fib` works! Commutativity of $+$ is not required.

Fibonacci with reducers

No restructuring:

```
cilk::reducer_opadd<int> x;

void fib(int n) {
  if (n < 2) {
    x += n;
  } else {
    cilk_spawn fib(n-1);
    fib(n-2);
  }
}

void doit(int n) {
  x.set_value(0);
  fib(n);
  printf("%d\n",
         x.get_value());
}
```

Fibonacci with reducers

No restructuring:

```
cilk::reducer_opadd<int> x;

void fib(int n) {
  if (n < 2) {
    x += n;
  } else {
    cilk_spawn fib(n-1);
    fib(n-2);
  }
}

void doit(int n) {
  x.set_value(0);
  fib(n);
  printf("%d\n",
         x.get_value());
}
```

This works too!

```
int cilk_main()
{
  for (int n = 0; i < N;
        ++n)
    cilk_spawn doit(n);
}
```

Reducers are not:

- Atomic objects.
- Transactional memory.
- Thread-local storage.
- #pragma omp private.

Uses and abuses of reducers

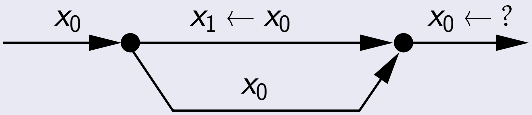
- Reductions over arbitrary algorithms and data structures.
  - E.g., collision detection.
- Parallel dynamic scoping.
  - E.g., multiple reductions in parallel.
- Implementation of exceptions.
- Reorder buffer in file I/O.
  - Parallelized bzip2 using a reducer.
  - Write the leftmost view to disk.
- Debugging/profiling.
  - Reduction of `int` over + with "identity" `1`: counts the number of views.
- Generalization of thread-local storage.
  - E.g., region-based memory allocator as a reducer.
  - Backward compatible with serial semantics.

Current status

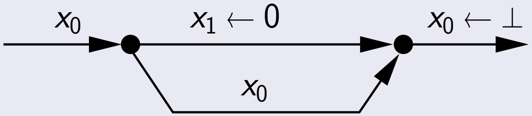
- Reducers are implemented as part of Cilk++.
- First-class objects. Can be stored in arrays, classes, etc.
- Not tied to loop, map, fold, or other control structures.
- Source code available at `www.cilk.com`.
- Efficient implementation with work-stealing scheduler. See [SPAA 2009].

Other possible hyperobjects

Splitters (hard to implement?):



Holders:



My thesis

- Parallelism is becoming a necessity.
- Stateful code is ubiquitous. (OO encourages stateful programming even when not needed, e.g. iterators.)
- Mutable state is (usually) incompatible with parallelism.
- Restructuring existing code is impractical.
- If you cannot change the program, find a suitable parallel meaning for the state.