Implicitly Parallel Programming with Ohua

Sebastian Ertel
Chair for Compiler Construction
TU Dresden
Explicit Parallel Programming

• Threads / Locks
• Message-passing / Channels
• Actors / Mailboxes

• Problems:
  • error-prone (data races, deadlocks)
  • no automatic scalability
  • hard to maintain because of code cluttering
What’s a hack?
The Problem with Threads

Edward A. Lee
Professor, Chair of EE, Associate Chair of EECS
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.
eal@eecs.berkeley.edu

January 10, 2006

Abstract

Threads are a seemingly straightforward adaptation of the dominant sequential model of computation to concurrent systems. Languages require little or no syntactic changes to support threads, and operating systems and architectures have evolved to efficiently support them. Many technologists are pushing for increased use of multithreading in software in order to take advantage of the predicted increases in parallelism in computer architectures. In this paper, I argue that this is not a good idea. Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism. Although many research techniques improve the model by offering more effective pruning, I argue that this is approaching the problem backwards. Rather than pruning nondeterminism, we should build from essentially deterministic, composable components. Nondeterminism should be explicitly and judiciously introduced where needed, rather than removed where not needed. The consequences of this principle are profound. I argue for the development of concurrent coordination languages based on sound, composable formalisms. I believe that such languages will yield much more reliable, and more concurrent programs.
Shared data and data races

```java
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while (i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 1: A Java implementation of the observer pattern, valid for one thread.
Deadlocks and Lock Composition

```java
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while (i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 1: A Java implementation of the observer pattern, valid for one thread.
Lost Updates

```java
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        List copyOfListeners;
        synchronized(this) {
            value = newValue;
            copyOfListeners = new LinkedList(listeners);
        }
        Iterator i = copyOfListeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 2: A Java implementation of the observer pattern that attempts to be thread safe.
These same computer vendors are advocating more multi-threaded programming, so that there is concurrency that can exploit the parallelism they would like to sell us. Intel, for example, has embarked on an active campaign to get leading computer science academic programs to put more emphasis on multi-threaded programming. If they are successful, and the next generation of programmers makes more intensive use of multithreading, then the next generation of computers will become nearly unusable.
Implicit Parallel Programming

• No additional abstractions, just functions/algorithms.

• The compiler introduces the parallelism.
Observations

*Algorithms* are built from *functionality*.

- Functionality: small units of code.
- Algorithm: composition of functions to implement a larger goal.

*State is mostly used locally.*

- Private local state.
- Shared global state.
Programming Model

\[
\text{program} = \text{algorithms} + \text{stateful functions}
\]

- parallelism
- state encapsulation
- \textit{declarative/functional}
- \textit{imperative/ OOP}
Programming Model

program = algorithms + stateful functions

parallelism

declarative/functional

Ohua = (Clojure + Java/Clojure/Scala)

state encapsulation

imperative/OOP
From Imperative to Declarative

• Example: Increment a list of values.

```
for (int i=0; i<v.length; i++)
r[i] = v[i]++
```

```
(map
  (fn [i] (+ i 1))
  v)
```
From Imperative to Declarative

• Example: Increment a list of values.

```python
for (int i=0; i<v.length; i++)
    r[i] = v[i]++
```

Think:

```
(map
    (fn [i] (+ i 1))
    v)
```
From Imperative to Declarative

- Example: Increment a list of values.

```python
for (int i=0; i<v.length; i++)
  r[i] = v[i]++

(map
  (fn [i] (+ i 1))
  v)
```

Think:
`lambda i: i+1`
From Imperative to Declarative

• Example: Increment a list of values.

```java
for (int i=0; i<v.length; i++)
    r[i] = v[i]++
```

• Map functions:

Think:
```
(map
    (fn [i] (+ i 1))
    v)
```

Diagram:
```plaintext
  map: (fn [i] (+ i 1))
   v1  v1'
   v2  v2'
   ...
   vn  vn'
```
From Imperative to Declarative

• Example: Increment a list of values.

```java
for (int i=0; i<v.length; i++)
    r[i] = v[i]++
```

• Map functions:

```java
(map (fn [i] (+ i 1)) v)
```

No side-effects!
From Imperative to Declarative

• Example: Increment a list of values.

```java
for (int i=0; i<v.length; i++)
    r[i] = v[i]++
```

• Map functions:

Map:

```
(map
    (fn [i] (+ i 1))
    v
)
```

Smap:

```
map:  

  v'    
  v'    
  i     
  v'    
  v'    
  i     
  v'    
  v'    
  i     
  v'    
  v'    
  i     
  v'    

S, v1
S, v2
...Sn
S(n-1), vn
S(n), vn

Side-effects to S.
```
Example: Key-value store

Algorithm:
Example: Key-value store

Algorithm:

```
(server-socket kv-map
  (smap
    (fn [cnn] ...
    ...))
```
Example: Key-value store

Algorithm:

```scheme
(smap
  (fn [cnn]
    (let [ [read-cnn req] (read cnn) ]
      server-socket kv-map
    )))
```
Example: Key-value store

Algorithm:

```clojure
(server-socket kv-map
  (smap
    (fn [cnn]
      (let [[read-cnn req] (read cnn)]
        (let [[req-type req-data] (parse req)]
          server-socket)))
```
Example: Key-value store

Algorithm:

```clojure
(smap
  (fn [cnn]
    (let [read-cnn [req] (read cnn)]
      (let [req-data [parse req]]
        (cond
          (= req-type "GET")
          (= req-type "PUT")
          {:server-socket kv-map}
        )
      )
    ))
```
Example: Key-value store

Algorithm:

``` Clojure
(server-socket kv-map
  (fn [cnn]
    (let [ [read-cnn req] (read cnn) ]
      (let [ [req-type req-data] (parse req) ]
        (cond
          (= req-type "GET") (parse-get req-data)
          (= req-type "PUT") .........)

  server-socket)
```
Example: Key-value store

Algorithm:

```
smap
  (fn [conn]
    (let [ [read-cnn req] (read cnn) ]
      (let [ [req-type req-data] (parse req) ]
        (cond
          (= req-type "GET") (get (parse-get req-data) kv-map)
          (= req-type "PUT") (assoc kv-map req-data)
          true server-socket)
    server-socket)
```
Example: Key-value store

Algorithm:

```clojure
(smap
(fn [cnn]
  (let [ [read-cnn req] (read cnn) ]
    (let [ [req-type req-data] (parse req) ]
      (cond
        (= req-type "GET") (get (parse-get req-data) kv-map)
        (= req-type "PUT") (parse-put req-data) 1)
    )
  server-socket)
```
Example: Key-value store

Algorithm:

```clojure
(smap
 (fn [cnn]
  (let [ [read-cnn req] (read cnn) ]
    (let [ [req-type req-data] (parse req) ]
      (cond
       (= req-type "GET") (get (parse-get req-data) kv-map)
       (= req-type "PUT") (put (parse-put req-data) kv-map)))
    server-socket)
```

server-socket kv-map
Example: Key-value store

Algorithm:

```clojure
(smap
 (fn [cnn]
  (let [ [read-cnn req] (read cnn) ]
   (let [ [req-type req-data] (parse req) ]
     (let [ value (cond
                    (= req-type "GET") (get (parse-get req-data) kv-map)
                    (= req-type "PUT") (put (parse-put req-data) kv-map)) ]
      server-socket)))
```
Example: Key-value store

Algorithm:

```
:smap
  (fn [cnn]
    (let [ [read-cnn req] (read cnn) ]
      (let [ [req-type req-data] (parse req) ]
        (let [ value (cond
            (= req-type "GET") (get (parse-get req-data) kv-map)
            (= req-type "PUT") (put (parse-put req-data) kv-map) ]
            (construct-response value)]]))
  server-socket)
```
Example: Key-value store

Algorithm:

```clojure
(smap
  (fn [cnn]
    (let [ [read-cnn req] (read cnn) ]
      (let [ [req-type req-data] (parse req) ]
        (let [ value (cond
                       (= req-type "GET") (get (parse-get req-data) kv-map)
                       (= req-type "PUT") (put (parse-put req-data) kv-map) ]
            (reply read-cnn (construct-response value)))))
   server-socket))
```
Example: Key-value store

Algorithm:

```clojure
(smap
 (fn [cnn]
  (let [ [read-cnn req] (read cnn) ]
   (let [ [req-type req-data] (parse req) ]
     (let [ value (cond
                   (= req-type "GET") (get (parse-get req-data) kv-map)
                   (= req-type "PUT") (put (parse-put req-data) kv-map)) ]
       (close (reply read-cnn (construct-response value))))))
 server-socket)
```
Example: Key-value store

Algorithm:

```clojure
(defn key-value-store [server-socket kv-map]
  (ohua
    (smap [cnn]
      (fn [read-cnn req] (read cnn))
      (let [ [req-type req-data] (parse req) ]
        (let [ value (cond
                      (= req-type "GET") (get (parse-get req-data) kv-map)
                      (= req-type "PUT") (put (parse-put req-data) kv-map)]]
          (close (reply read-cnn (construct-response value)))))
server-socket)))
```
Example: Key-value store

Stateful functions:

Algorithm:

```clojure
(ohua :import com.key.value.store)
(defn key-value-store [server-socket kv-map]
  (ohua
   (smap
    (fn [cnn]
      (let [ [read-cnn req] (read cnn) ]
        (let [ [req-type req-data] (parse req) ]
          (let [ value (cond
                        (= req-type "GET") (get (parse-get req-data) kv-map)
                        (= req-type "PUT") (put (parse-put req-data) kv-map) ]
            (close (reply read-cnn (construct-response value))))]
            server-socket))))
```
Example: Key-value store

Stateful functions:

Algorithm:

```plaintext
(ohua :import com.key.value.store)

(defn key-value-store [server-socket kv-map]
  (ohua
    (smap
      (fn [cnn]
        (let [[read-cnn req] (read cnn)]
          (let [[req-type req-data] (parse req)]
            (let [value (cond
                          (= req-type "GET") (get (parse-get req-data) kv-map)
                          (= req-type "PUT") (put (parse-put req-data) kv-map)]
              (close (reply read-cnn (construct-response value)))))
            (server-socket))))

package com.key.value.store;

public class GetAccess {
  @Function public Object get(Object key, Map<Object,Object> kvStore) {
    return kvStore.get(key); }
}
Example: Key-value store

Stateful functions:

Algorithm:
State

(ohua :import com.key.value.store)

(defn key-value-store [server-socket kv-map]
  (ohua
    (smap
      (fn [cnn]
        (let [ [read-cnn req] (read cnn) ]
          (let [ [req-type req-data] (parse req) ]
            (let [ value (cond
                          (= req-type "GET") (get (parse-get req-data) kv-map)
                          (= req-type "PUT") (put (parse-put req-data) kv-map)) ]
              (close (reply read-cnn (construct-response value))))))
          server-socket))))
State

Algorithm state

```
(ohua :import com.key.value.store)
(defn key-value-store [server-socket kv-map]
  (ohua
    (smap
      (fn [cn]
        (let [[read-cnn req] (read cnn)]
          (let [[req-type req-data] (parse req)]
            (let [value (cond
                          (= req-type "GET") (get (parse-get req-data) kv-map)
                          (= req-type "PUT") (put (parse-put req-data) kv-map)])]
              (close (reply read-cnn (construct-response value))))))
  server-socket))```

State

Function call state
State

- State per call-site.
- State is preserved in between invocations (throughout computation).
Ohua Infrastructure

- Embedded Domain Specific Language (EDSL)

(ohua expression)
Ohua Infrastructure

Embedded Domain Specific Language (EDSL)

Expression Language

\[ t ::= v \]

\[ | (\text{algo } [v] t) \]

\[ | (t t) \]

\[ | (\text{let } [v t] t) \]

\[ | (\text{if } t t t) \]

\[ | (\text{sf } t) \]

\[ (\text{smap } (\text{algo } [v] t) [v_1 \ldots v_n]) \]
Ohua Infrastructure

Embedded Domain Specific Language (EDSL)

Expression Language

Dataflow Language

Expression Language:
\[ t ::= v \mid (\text{algo } [v] \ t) \mid (t \ t) \mid (\text{let } [v \ t \ t] \ t) \mid (\text{if } t \ t \ t) \mid (\text{sf } t) \]

Dataflow Language:
\[ t ::= v \mid (\text{let } [[v_1 \ldots v_m] (\text{sf } v_1 \ldots v_n)] \ t) \mid (\text{let } [v (\text{sf } v_1 \ldots v_n)] \ t) \mid (\text{if } v \ v \ v) \]

\[ (\text{smap } (\text{algo } [v] \ t) [v_1 \ldots v_n]) \]
Ohua Infrastructure

Embedded Domain Specific Language (EDSL)

Expression Language

Dataflow Language

Dataflow Runtime

\( t ::= \) (ohua expression)

\( t ::= v \)
\( | (\text{algo } [v]\ t) \)
\( | (t\ t) \)
\( | (\text{let } [v\ t]\ t) \)
\( | (\text{if } v\ t\ t) \)
\( | (\text{smap } (\text{algo } [v]\ t)\ [v_1 \ldots v_n]) \)

\( t ::= \) (arc)
\( | (\text{port}) \)
\( | (\text{node}) \)

\( \text{threads/actors/…} \)
\( \text{queues/channels/…} \)
\( \text{scheduler} \)

\( (\text{smap } (\text{algo } [v]\ t)\ [v_1 \ldots v_n]) \)
Let’s get our hands dirty!

• Download an Ohua workspace from here:
  • https://github.com/ohua-dev/ir-transformation-workspace

• Make sure you have Java, Clojure and Leiningen installed.

• Make yourself comfortable with compilation and execution.
Your First Ohua Application

- Implement a program for audio file manipulation:
  1. Read in an audio (wav) file and split the two channels.
  2. Perform FFT
  3. Apply a bandpass filter to reduce noise.
  4. Perform iFFT
  5. Write file back to disk.

- Libraries are already present in the workspace.

- Version 1: without `smap` —> Version 2: with `smap`