# A Deterministic Multi-Way Rendezvous Library for Haskell

Nalini Vasudevan
Columbia University

Satnam Singh
Microsoft Research

Stephen A. Edwards
Columbia University

# Motivation

- In general, concurrent programming languages are non-deterministic
- Example of non-determinism in C

```
int x = 1;
void* bar(void* args) {
        x = x*2;
}
void foo() {
        pthread_create(&thread, NULL, bar, NULL);
        x++;
        printf("%d", x);
}
```

# Motivation

- Ensure atomicity by using locks

```
int x = 1;
void* bar(void* args) {
        pthread_mutex_lock(&mutex);
        x = x*2;
        pthread_mutex_unlock(&mutex);
}
void foo() {
        pthread_create(&thread, NULL, bar, NULL);
        pthread_mutex_lock(&mutex);
        x++;
        pthread_mutex_unlock(&mutex);
        printf("%d", x);
}
```

# Motivation

```
int x = 1;
void* bar(void* args) {
        pthread_mutex_lock(&mutex);
        x = x*2;
        pthread_mutex_unlock(&mutex);
}
void foo() {
        pthread_create(&thread, NULL, bar, NULL);
        pthread_mutex_lock(&mutex);
        x++;
        pthread_mutex_unlock(&mutex);
        printf("%d", x);
}
```

Output: 3 or 4

| | |
|---|---|
| x = 1 | x = 1 |
| bar: x = 1 * 2 = 2 | foo: x = 2 |
| foo: x = 3 | bar: x = 2*2 = 4 |

# Solution

- A deterministic communication library
- Easy to program and debug
- Speed-up without much compromise on determinism

# Solution

We chose Haskell as the language

Concurrency in Haskell:

- *forkIO*: Sparks off a new thread.
- *putMVar*: Puts a value into an MVar if empty, else blocks
- *takeMVar*: Removes and returns the contents of the MVar if it was full, else blocks

```
m <- newEmptyMVar --Create a new mailbox
forkIO (putMVar m (5::Int)) --Thread writes 5 to m
result <- takeMVar m
```

# Non-determinism in Haskell

$m <- newEmptyMVar$ -- Create a new mailbox

$forkIO\ (putMVar\ m\ (3::\textbf{Int}))$ -- thread writes 3 to m

$forkIO\ (putMVar\ m\ (4::\textbf{Int}))$ -- thread writes 4 to m

$a <- takeMVar\ m$ -- parent thread reads m

$b <- takeMVar\ m$ -- parent thread reads m

$\textbf{putStrLn}\ (\textbf{show}\ a)$

Output: 3 or 4

# Our Approach

Uses the SHIM model:

- Parent thread waits for children to finish
- Threads run asynchronously but synchronize (communicate) when data has to be shared
- Multi-way rendezvous style of communication

# Our Library's API

## Producer-Consumer Example

```
produce [c]
  = do
      val <- produceData
      dSend c val
      if val == -1 then --End of data
           return ()
        else
           produce [c]
```

```
consume [c]
  = do
      val <- dRecv c
      if val == -1 then --End of data
          return ()
        else
          do consumeData val
             consume [c]
```

```
producerConsumer
  = do
      c <- newChannel
      (_,_) <- dPar produce [c]
                    consume [c]
      return ()
```

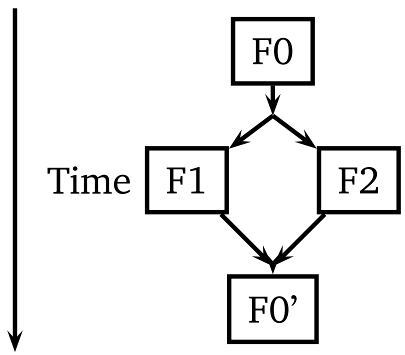
# Comparisons

| **Senders** | **Receivers** | **Mailboxes** | **Our Library** |
|---|---|---|---|
| 1 | 1 | Deterministic | Deterministic |
| 1 | $> 1$ | Nondeterministic | Deterministic |
| $> 1$ | – | Nondeterministic | Run-time error |

# Concurrency

- *DPar* example

$$(r1, r2) <- dPar\ func1\ clist1\ func2\ clist2$$




- Statement after *dPar* executed only after *dPar* returns
- Implementation
  - Either
    - Fork two threads for F1 and F2
    - Fork a thread for F1 and parent executes F2
  - Updates channel connection information

# Communication Protocol

Sender

1. Wait for a ready signal from every receiver
2. Send data to every receiver
3. Signal every receiver to proceed

Receiver

1. Send a ready signal to the sender
2. Receive data from sender
3. Wait for a proceed signal from sender

# Multi-way Rendezvous

**Thread 1**

- dSend c val
- ..
- ..
- dRecv c
- ..
- ..
- ..
- ..
- dRecv c

**Thread 2**

- dRecv c
- ..
- ..
- ..
- ..
- dRecv c
- ..
- dSend c val
- ..

**Thread 3**

- ..
- dRecv c
- ..
- ..
- ..
- dSend c val
- ..
- ..
- dRecv c

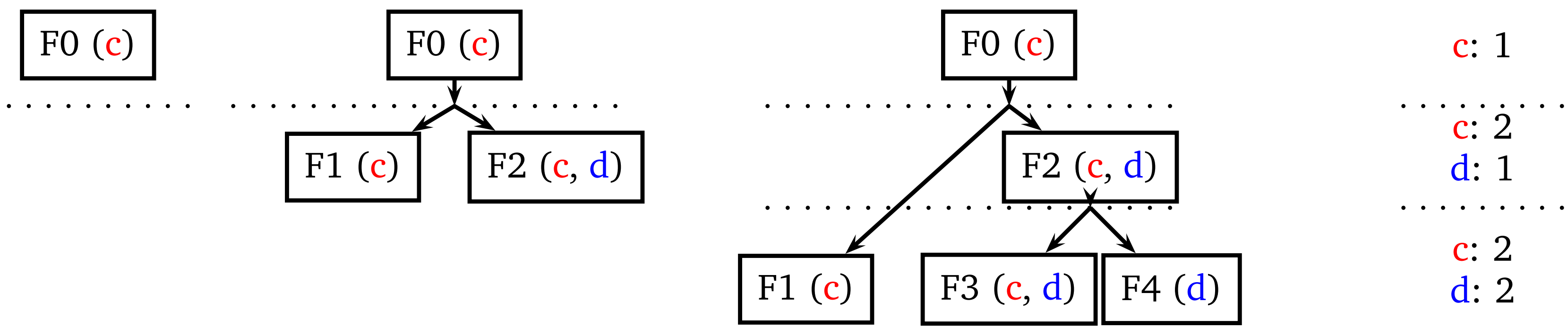
# Channel Connections

```
F0                              F2 [c]
  = do                            = do
      ..                              ..
      c <- newChannel                 d <- newChannel
      dPar F1 [c] F2 [c]              dPar F3 [c,d] F4 [d]
      ..                              ..
```

- F0 is connected to *c*
- F1 is connected to *c*
- F2 is connected to *c, d*
- F3 is connected to *c, d*
- F4 is connected to *d*

# Channel Connections

- F0 is connected to *c*
- F1 is connected to *c*
- F2 is connected to *c*,*d*
- F3 is connected to *c*,*d*
- F4 is connected to *d*




Initial state | F0 calls *dPar F1 [c] F2 [c]* | F2 calls *dPar F3 [c,d] F4 [d]* | Connections

# Two Implementations

1. Mailboxes

   $m <- newEmptyMVar$ −− Create a new mailbox

   $forkIO\ (putMVar\ m\ (5{::}\textbf{Int}))$ −− Thread writes 5 to m

   $result <- takeMVar\ m$

2. Software Transactional Memory (STM)

   - Lock-free implementation
   - Thread completes modification to shared memory without regard for other threads
   - Threads are validated before commit
   - If conflict, roll-back.

# Two Implementations

- *atomically*: STM action
- *retry*: Blocking action

```
atomically (do
        value <- readTVar c
        if value == -1 then
          retry -- Not written yet
         else writeTVar c (value + 1))
```

# STM Versus Mailboxes

- Implemented the library using both methods
- Made *N* threads, rendezvous once
- Only communication, no computation
- Experimented on 1.6 GHz Intel Core 2 Duo, 500 MB RAM, Windows XP machine

# STM Versus Mailboxes

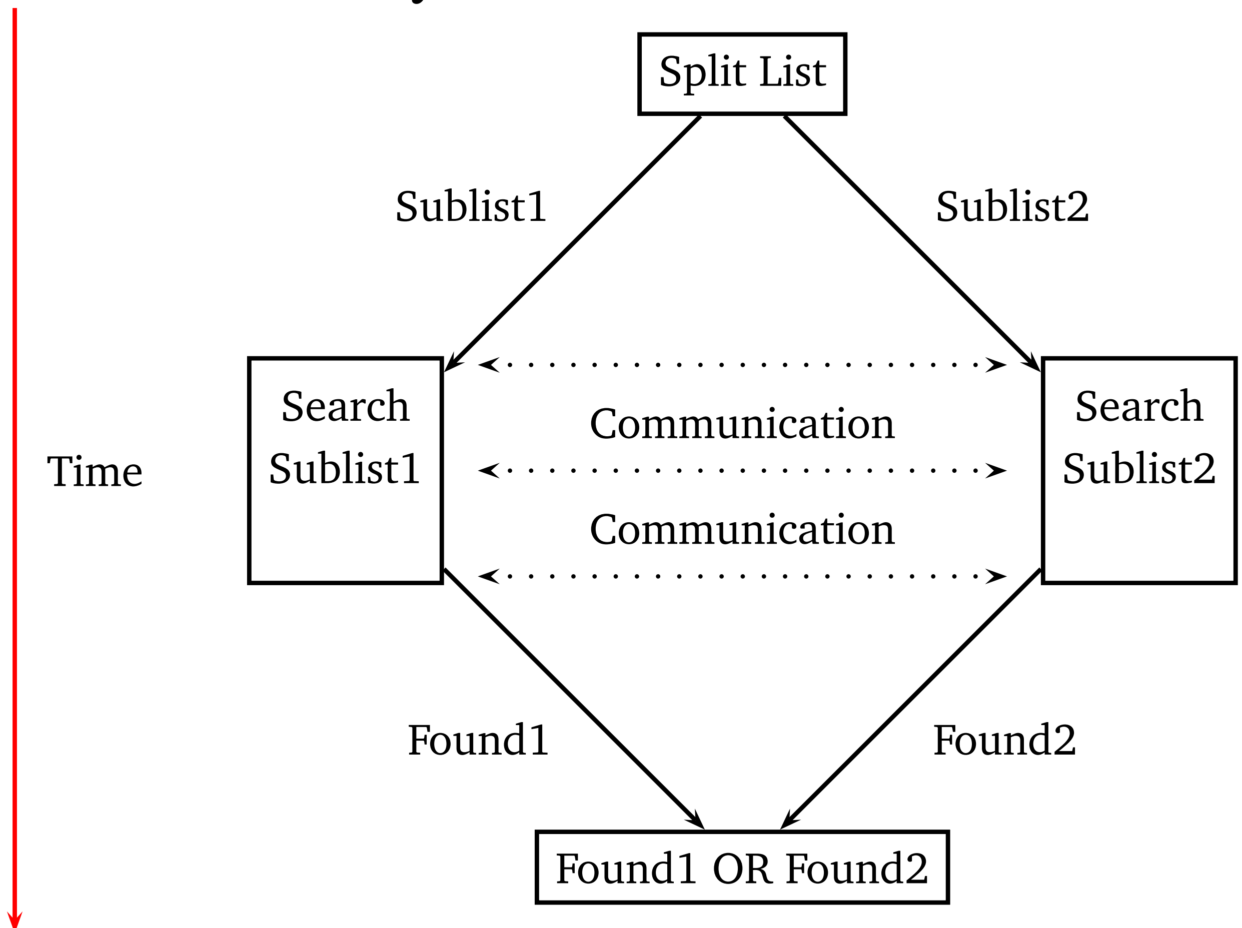
| Threads | Time to Rendezvous | | Speedup |
|---|---|---|---|
| | STM | Mailbox | (STM/Mailbox) |
| 2 | 0.11 ms | 0.07 ms | 1.6 |
| 3 | 0.14 | 0.08 | 1.8 |
| 4 | 0.17 | 0.14 | 1.2 |
| 5 | 0.21 | 0.16 | 1.3 |
| 6 | 0.28 | 0.17 | 1.6 |
| 7 | 0.31 | 0.21 | 1.5 |
| 8 | 0.37 | 0.23 | 1.6 |
| 9 | 0.42 | 0.27 | 1.6 |
| 10 | 0.47 | 0.28 | 1.7 |
| 100 | 6.4 | 1.8 | 3.5 |
| 200 | 35 | 6.7 | 5.2 |
| 400 | 110 | 14 | 7.7 |
| 800 | 300 | 34 | 8.9 |

# More Experiments

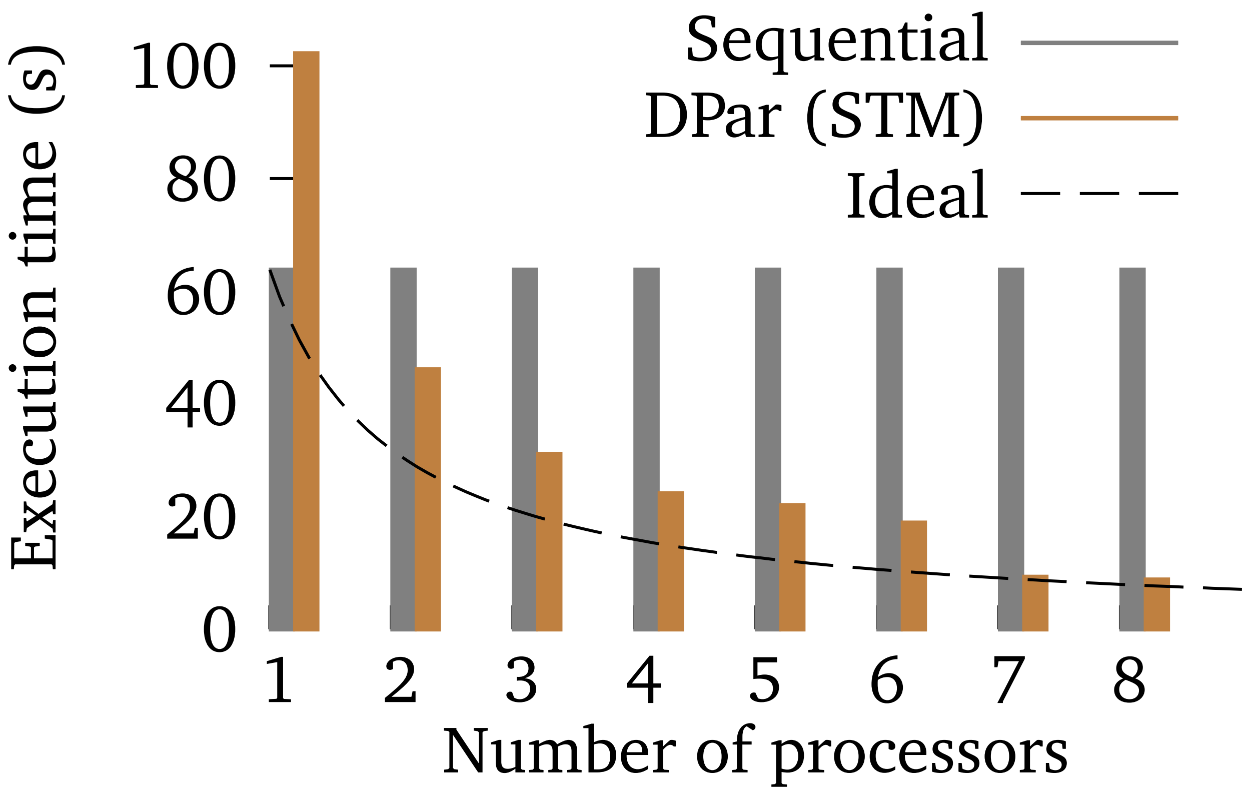
- Experimented on 8-processor, two 1.6 GHz quad core, 2 GB RAM, Windows NT server
- Applications
  - Maximum Finder
  - SAT Solver
  - Linear Search
  - Systolic Filter
  - RGB Histogram

# Linear Search

- Rendezvous at regular intervals
- N = 420 000 and key at position 390 000
- Sub-tasks synchronize 5 times

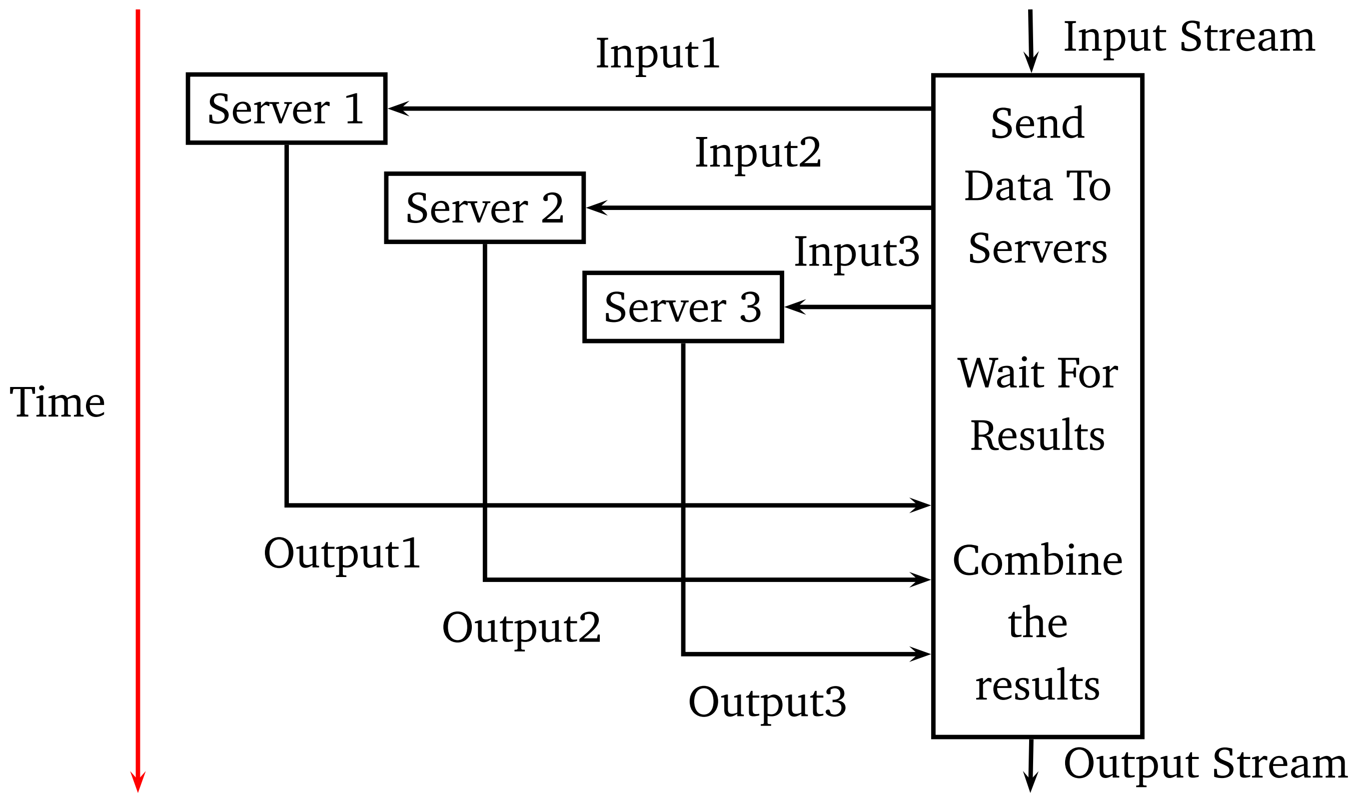


# Linear Search

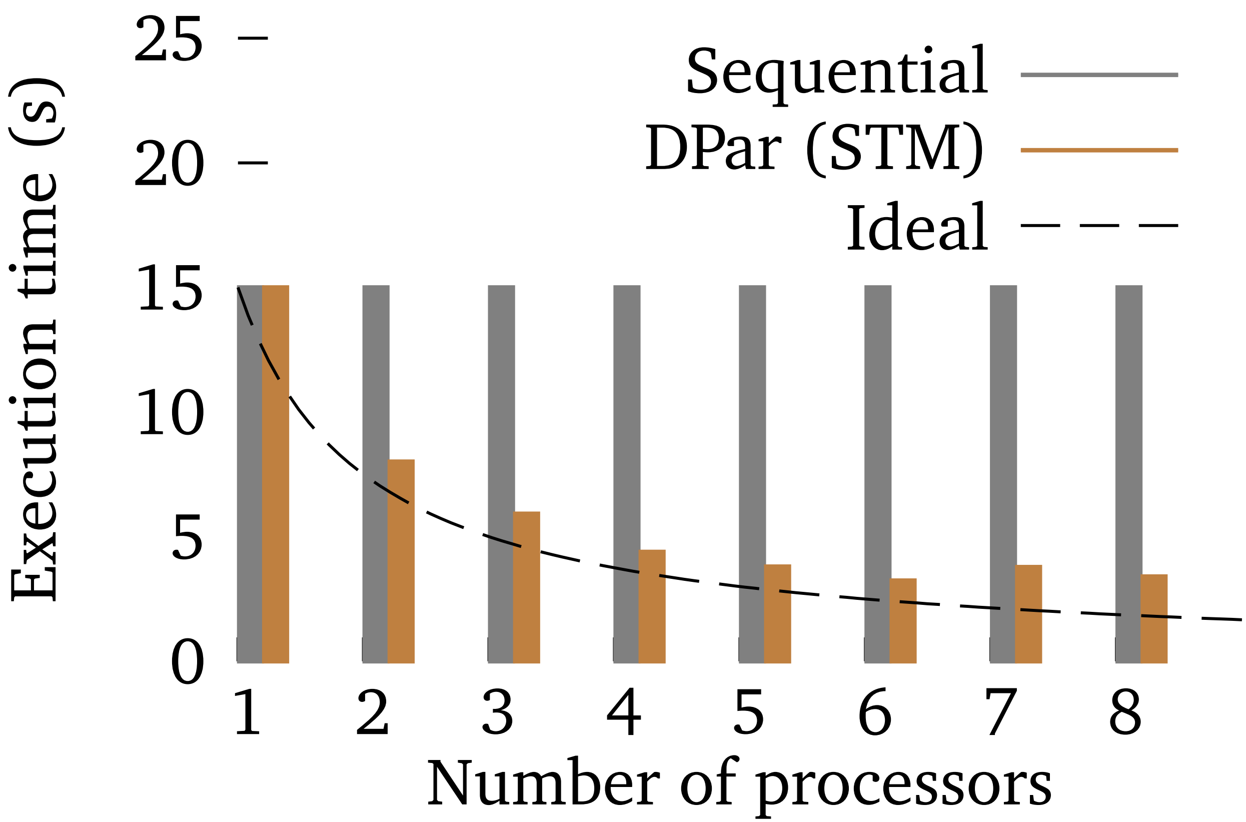


# RGB Histogram

- 560KB Raster file
- Client Server Model
- Other Examples: Systolic Filter, FFT



# RGB Histogram



# Conclusions

- Easy, less error-prone programming model
- Compared STM and Mailbox implementations
- Efficient: Considerable speed-up
- Future Work: Deterministic concurrent exceptions