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

```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

```c
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

```c
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
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:

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

\begin{verbatim}
m <- newEmptyMVar  -- Create a new mailbox
forkIO (putMVar m (5::Int))  -- Thread writes 5 to m
result <- takeMVar m
\end{verbatim}
Non-determinism in Haskell

\[
m \leftarrow \text{newEmptyMVar} \quad \text{-- Create a new mailbox}
\]

\[
\text{forkIO} \ (\text{putMVar} \ m \ (3::\text{Int})) \quad \text{-- thread writes 3 to } m
\]

\[
\text{forkIO} \ (\text{putMVar} \ m \ (4::\text{Int})) \quad \text{-- thread writes 4 to } m
\]

\[
a \leftarrow \text{takeMVar} \ m \quad \text{-- parent thread reads } m
\]

\[
b \leftarrow \text{takeMVar} \ m \quad \text{-- parent thread reads } m
\]

\[
\text{putStrLn} \ (\text{show} \ a) \quad \text{Output: 3 or 4}
\]
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
Producer-ConSUMER Example

\[
\text{produce} \ [c] = \text{do} \\
\quad \text{val} \leftarrow \text{produceData} \\
\quad \text{dSend} \ c \ \text{val} \\
\quad \text{if} \ \text{val} == -1 \text{ then } \text{--End of data} \\
\quad \quad \quad \text{return} () \\
\quad \text{else} \\
\quad \quad \quad \text{produce} \ [c]
\]

\[
\text{consume} \ [c] = \text{do} \\
\quad \text{val} \leftarrow \text{dRecv} \ c \\
\quad \text{if} \ \text{val} == -1 \text{ then } \text{--End of data} \\
\quad \quad \quad \text{return} () \\
\quad \text{else} \\
\quad \quad \quad \text{do} \ \text{consumeData} \ \text{val} \\
\quad \quad \quad \text{consume} \ [c]
\]

\[
\text{producerConsumer} = \text{do} \\
\quad \text{c} \leftarrow \text{newChannel} \\
\quad (\_,\_) \leftarrow \text{dPar} \ \text{produce} \ [c] \\
\quad \ \text{consume} \ [c] \\
\quad \text{return} ()
\]
## Comparisons

<table>
<thead>
<tr>
<th>Senders</th>
<th>Receivers</th>
<th>Mailboxes</th>
<th>Our Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Deterministic</td>
<td>Deterministic</td>
</tr>
<tr>
<td>1</td>
<td>&gt; 1</td>
<td>Nondeterministic</td>
<td>Deterministic</td>
</tr>
<tr>
<td>&gt; 1</td>
<td>–</td>
<td>Nondeterministic</td>
<td>Run-time error</td>
</tr>
</tbody>
</table>
Concurrency

- *DPar* example

\[(r_1, r_2) \leftarrow dPar \text{ func1 clist1 func2 clist2}\]

- 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

<table>
<thead>
<tr>
<th><strong>Sender</strong></th>
<th><strong>Receiver</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Wait for a ready signal from every receiver</td>
<td>1. Send a ready signal to the sender</td>
</tr>
<tr>
<td>2. Send data to every receiver</td>
<td>2. Receive data from sender</td>
</tr>
<tr>
<td>3. Signal every receiver to proceed</td>
<td>3. Wait for a proceed signal from sender</td>
</tr>
</tbody>
</table>
Multi-way Rendezvous

Thread 1
- dSend c val
- ...
- ...
- dRecv c
- ...
- ...
- ...
- dRecv c

Thread 2
- dRecv c
- ...
- ...
- ...
- ...
- ...
- dRecv c
- ...
- dSend c val
- ...
- ...
- dRecv c

Thread 3
- ...
- ...
- ...
- ...
- ...
- ...
- dRecv c
- ...
- dSend c val
- ...
- ...
- dRecv c
Channel Connections

\[ F0 = do \]
\[ \quad \ldots \]
\[ \quad c \leftarrow \text{newChannel} \]
\[ \quad \text{dPar } F1 \ [c] F2 \ [c] \]
\[ \quad \ldots \]
\[ \quad \text{F0 is connected to } c \]
\[ \quad \text{F1 is connected to } c \]
\[ \quad \text{F2 is connected to } c, d \]
\[ \quad \text{F3 is connected to } c, d \]
\[ \quad \text{F4 is connected to } d \]

\[ F2 \ [c] = do \]
\[ \quad \ldots \]
\[ \quad d \leftarrow \text{newChannel} \]
\[ \quad \text{dPar } F3 \ [c,d] F4 \ [d] \]
\[ \quad \ldots \]
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

```
F0 (c)  F0 (c)
       |       |
     F1 (c) F2 (c, d)
```

```
F0 (c)
       |       |
     F2 (c, d)
       |       |
     F1 (c) F3 (c, d) F4 (d)
```

```
c: 1
  c: 2
d: 1
c: 2
d: 2
```
Two Implementations

1. Mailboxes

   \[ m \leftarrow \text{newEmptyMVar} \quad \text{-- Create a new mailbox} \]
   \[ \text{forkIO (putMVar } m \ (5::\text{Int}) \text{)} \quad \text{-- Thread writes 5 to } m \]
   \[ \text{result} \leftarrow \text{takeMVar } m \]

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

\[
\text{atomically (do}
\begin{align*}
\text{value } & \leftarrow \text{readTVar } c \\
\text{if value } & = -1 \text{ then }
\begin{align*}
\text{retry} & \quad \text{Not written yet}
\end{align*}
\text{else writeTVar } c \ (\text{value } + 1))
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time to Rendezvous</th>
<th>Speedup (STM-Mailbox)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STM</td>
<td>Mailbox</td>
</tr>
<tr>
<td>2</td>
<td>0.11 ms</td>
<td>0.07 ms</td>
</tr>
<tr>
<td>3</td>
<td>0.14</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>0.17</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>0.21</td>
<td>0.16</td>
</tr>
<tr>
<td>6</td>
<td>0.28</td>
<td>0.17</td>
</tr>
<tr>
<td>7</td>
<td>0.31</td>
<td>0.21</td>
</tr>
<tr>
<td>8</td>
<td>0.37</td>
<td>0.23</td>
</tr>
<tr>
<td>9</td>
<td>0.42</td>
<td>0.27</td>
</tr>
<tr>
<td>10</td>
<td>0.47</td>
<td>0.28</td>
</tr>
<tr>
<td>100</td>
<td>6.4</td>
<td>1.8</td>
</tr>
<tr>
<td>200</td>
<td>35</td>
<td>6.7</td>
</tr>
<tr>
<td>400</td>
<td>110</td>
<td>14</td>
</tr>
<tr>
<td>800</td>
<td>300</td>
<td>34</td>
</tr>
</tbody>
</table>
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

Execution time (s)

Number of processors

Sequential
DPar (STM)
Ideal

1 2 3 4 5 6 7 8

0 20 40 60 80 100
RGB Histogram

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

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