Graph Algorithms in a Guaranteed-Deterministic Language

Praveen Narayanan & Ryan R. Newton
Outline

- An example of non-determinism
- Introduction to LVish
- BFS and MIS in LVish
- BulkRetry, a new addition
Graph algorithms

- *Irregular* parallel behavior
- Amount of parallelism depends on input
Parallel BFS

Level 0
Parallel BFS

Level 0
Parallel BFS
Parallel BFS

Level 1

\[ r \rightarrow w \]
Parallel BFS
Parallel BFS

Level 1

Level 2
Parallel BFS

Level 2

Level 1
Parallel BFS

Level 2
Parallel BFS

Level 2

Level 3
Parallel BFS

![Graph](image)

- **Level 2**: \( v \), \( w \), \( x \)
- **Level 3**: \( u \)
Parallel BFS

Level 2

Level 3
Parallel BFS
Parallel BFS
Parallel BFS
Choose a different winner
Different BFS tree

Level 2

Level 3
Different BFS tree

![BFS Tree Diagram]

Level 2

Level 3
Different BFS tree
Different BFS tree

- Different BFS tree diagram with nodes labeled as follows:
  - Node 1 connected to nodes 2 and 0
  - Node 2 connected to nodes 1 and 3
  - Node 3 connected to nodes 2 and 0

- Nodes u, v, w, x, y
- Level 3
Different BFS tree

Level 3
Non-determinism
How could we achieve determinism?

- Carefully write code in a *non-deterministic* language [Blelloch et al., 2012]

- *Program-level* guarantee only

  \[ \text{node.visited} = \text{true} \]
How could we achieve determinism?

- Implement algorithms in a guaranteed-deterministic language (this talk)

- Language-level guarantee
LVars

• Lattice Variables [FHPC ’13]

• Monotonic data structures
LVars

• Multiple least-upper-bound writes

Lattice for `num`

Raises an error, since $3 \sqcup 4 = \top$

```
do
    fork (put num 3)
    fork (put num 4)
```

Fine, since $4 \sqcup 4 = 4$

```
do
    fork (put num 4)
    fork (put num 4)
```
LVars

• Blocking threshold reads, i.e., \( \text{get} \)

\[
\text{let par } _ = \text{put lv } (_,4) \\
_ = \text{put lv } (3,_) \\
x = \text{getSnd lv} \\
\text{in } x
\]

• See intermediate state of the data structure
LVish

• A Haskell library for programming with LVars

• POPL ’14, PLDI ’14
Handlers in LVish

- Callback functions
- Added to an LVar (using `addHandler`)
- Spawned every time an LVar changes
connected component

connected :: Graph → NodeID → Par (ISet NodeID)
connected g startV = do
    seen ← newEmptySet
    addHandler seen
    (λ nd → parMapM (putInSet seen) (nbrs g nd))
    putInSet seen startV
    return seen
Connected component
Connected component

putInSet seen startV
Connected component

\[ \lambda \text{nd} \to \text{parMapM} \ (\text{putInSet} \ \text{seen}) \ (\text{nbrs} \ g \ \text{nd}) \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} \left( \text{putInSet} \ \text{seen} \right) \left( \text{nbrs} \ \text{g} \ \text{nd} \right) \]
Connected component

\[ \lambda \ nd \rightarrow \ parMapM \ (putInSet \ seen) \ (nbrs \ g \ nd) \]

\[ u \ v \ t \ t \]
Connected component

\[ \text{\textlambda } nd \rightarrow \text{parMapM (putInSet seen) (nbrs g nd)} \]

\[
\begin{array}{cccc}
u & u & v & t \\
\end{array}
\]
Connected component

\[ \lambda \text{nd} \to \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component

\[ \lambda \text{nd} \to \text{parMapM (putInSet seen) (nbrs g nd)} \]

\[ u \quad v \quad x \quad t \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM (putInSet seen) (nbrs g nd)} \]
Connected component

\[ \lambda \text{nd} \to \text{parMapM (putInSet seen) (nbrs g nd)} \]

\[ \text{u} \quad \text{x} \quad \text{t} \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} \ (\text{putInSet} \ \text{seen}) \ (\text{nbrs} \ g \ \text{nd}) \]
Connected component

\[ \lambda \ nd \rightarrow \ \text{parMapM} \ (\text{putInSet} \ seen) \ (\text{nbrs} \ g \ nd) \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component

\[\lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd})\]

\[
\begin{align*}
\text{s} & \rightarrow \text{v} \\
\text{r} & \rightarrow \text{u} \\
\text{t} & \rightarrow \text{w} \\
\text{x} & \rightarrow \text{y} \\
\text{z} & \rightarrow \text{y}
\end{align*}
\]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM (putInSet seen) (nbrs g nd)} \]

\[ v \quad y \quad t \quad t \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM (putInSet seen) (nbrs g nd)} \]
Connected component

\( \lambda \text{nd} \rightarrow \text{parMapM (putInSet seen) (nbrs g nd)} \)
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component

\[ \lambda \text{nd} \to \text{parMapM (putInSet seen) (nbrs g nd)} \]
Connected component

\[ \lambda \ nd \rightarrow \ parMapM \ (putInSet \ seen) \ (nbrs \ g \ nd) \]
Connected component

\[ \lambda \text{nd} \rightarrow \text{parMapM} (\text{putInSet seen}) (\text{nbrs g nd}) \]
Connected component
Breadth-first search

- Similar to connected
- Return BFS tree topology rather than “seen” set
Deterministic spanning tree

- Use a reduction variable to track highest priority parent
- $MinVar$, an instance of LVars
import Data.LVar.MinVar

type NodeID = Int

bfsTree :: Graph → NodeID → Par (Vector MinVar)
bfsTree gr start = do
    let (Graph verts edges) = gr
    parents ← Vector.generateM (length verts) (λ ix → newMinVar maxInt)
    seen ← newEmptySet
    let handler nd = mapM (eachNbr nd) (nbrs g nd)
        eachNbr nd nbr = do
            putInSet seen nbr
            putMin parents nbr nd
    addHandler seen handler -- Register callback.
    putInSet seen start -- Kick things off.
return parents
Deterministic tree
Deterministic tree

Vector.generateM (length verts) (λ ix → newMinVar maxInt)
Deterministic tree

putInSet seen start

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
<td>maxInt</td>
</tr>
</tbody>
</table>
Deterministic tree

do putInSet seen nbr
putMin parents nbr nd
Deterministic tree

1  2  3

4  5  6

7  8  9

maxInt | 1  | 1  | maxInt | maxInt | maxInt | maxInt | maxInt | maxInt | maxInt

1  2  3  4  5  6  7  8  9
Deterministic tree

```
maxInt
maxInt
maxInt
maxInt
maxInt
maxInt
maxInt
maxInt

1 2 3 4 5 6 7 8 9
```

```
do putInSet seen nbr
putMin parents nbr nd
```
Deterministic tree

<table>
<thead>
<tr>
<th>maxInt</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>maxInt</th>
<th>maxInt</th>
<th>maxInt</th>
<th>maxInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Deterministic tree

<table>
<thead>
<tr>
<th>maxInt</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>7</th>
<th>4</th>
<th>8</th>
<th>maxInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Deterministic tree

```
maxInt 1 1 2 2 7 4 8 maxInt
1 2 3 4 5 6 7 8 9
```
Rest of the talk

- Second application of LVish to graphs
- Motivation for improvement
- BulkRetry
Maximal independent set

- Spawn a task for each vertex
- Check if any *higher-priority* neighbor has been added
Maximal independent set

• Priorities handled using the blocking \texttt{get}

\texttt{get flagsArr nbr}

\texttt{data Flag = Chosen | NbrChosen}
Maximal independent set
Maximal independent set
Maximal independent set

1. \{\}
2. \{1\}
3. \{2\}
4. \{2\}
5. \{\}
6. \{3,5\}
7. \{2,4\}
8. \{2,6,7\}
Maximal independent set
Maximal independent set
Maximal independent set

```
{2}  
{2}  
{3,5} {2,4} {2,6,7}
```
Maximal independent set

\begin{itemize}
\item \{3,5\}
\item \{2,4\}
\item \{2,6,7\}
\end{itemize}
Maximal independent set

{2, 6, 7}
Maximal independent set
Observe

- Inefficient for large $|V|$ 
- Per vertex interaction with task scheduler
Deterministic Reservations

- Loop carried dependencies

- “Internally Deterministic Parallel Algorithms Can Be Fast”, Blelloch et al., ’12
Deterministic Reservations
Deterministic Reservations

Round 1
Deterministic Reservations

Round 1
Reserve Phase
Deterministic Reservations

Round 1

Commit Phase
Deterministic Reservations

Round 2
Deterministic Reservations

Round 2

Reserve Phase
Deterministic Reservations

Round 2

Commit Phase
Deterministic Reservations

Round 3
Deterministic Reservations

Round 3

Reserve Phase
Deterministic Reservations

Round 3

Commit Phase
Deterministic Reservations

Round 4
Deterministic Reservations

Round 4

Reserve Phase
Deterministic Reservations

Round 4

Commit Phase
Deterministic Reservations

Round 5
Deterministic Reservations

Round 5

Reserve Phase
Deterministic Reservations

Round 5

Commit Phase
Deterministic Reservations

Lessons:
- Bulk processing
- Retry rather than blocking
BulkRetry in LVish

- Bulk: For \( n = pq \) iterations, spawn \( p \) tasks, each processing \( q \) iterations in sequence

- If the iteration dependency is not met, abort and retry later

- LVish operations are *idempotent*: retry and blocking have same semantics
BulkRetry in LVish
BulkRetry in LVish

Parallel task 1
1 2 3 4

Parallel task 2
5 6 7 8
BulkRetry in LVish

Parallel task 1
1
2
3
4

Parallel task 2
5
6
7
8
BulkRetry in LVish

Parallel task 1
1
2
3
4

Parallel task 2
5
6
7
8
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

2
3
4

Parallel task 2

6
7
8
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish
 BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 1

Parallel task 2
BulkRetry in LVish

Parallel task 2
BulkRetry in LVish

Parallel task 2
BulkRetry in LVish
Evaluation

- Consistent speedup compared to vanilla LVish MIS
- LVish MIS gains 3x - 10x speedup
- 10x slower than PBBS (C++) implementations
Future work

- Randomized scheduler
- Constant-factor optimizations
- Improved data structures
Conclusion

• Compared approaches that deterministically process graphs

• Thank you!