“They [threads] discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads . . . are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism.”

Outline of the paper

• Very brief introduction to concurrent programming
• Shows the “essence” of the problem w/ threads from a computational viewpoint
• Illustrates the practical implications of the problem via
  - A simple, but useful example (observer pattern)
  - An anecdote from a real project using threads (see paper)
• Lightly surveys aggressive pruning techniques
• Illustrates and argues for an alternative to threads & pruning: concurrent coordination languages

Introduction

• Why bother with concurrent programming?
  – Take advantage of parallel architectures
  – The solution to the end of Moore’s law
• Domains for which threads are not used
  – Scientific computing
    ◦ Data parallel language extensions
    ◦ Message passing libraries (PVM, MPI, OpenMP)
  – Distributed computing
    ◦ But mechanisms like CORBA and .NET make distributed programming look like multithreaded programming ...
  – Embedded computing
    ◦ Special architectures (combine SIMD, VLIW, stream processing)
    ◦ Assembly code combined with C code
  “... achieving reliability and predictability using threads is essentially impossible for many applications.”

Computational model of threads

Notation:

<table>
<thead>
<tr>
<th>$B = {0, 1}$</th>
<th>binary digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{N} = {0, 1, 2, ...}$</td>
<td>natural numbers</td>
</tr>
<tr>
<td>$B^\alpha = B \cup B^\omega$</td>
<td>sequences of 0’s and 1’s</td>
</tr>
<tr>
<td>$B^* \rightarrow B^*$</td>
<td>partial functions over $B^*$</td>
</tr>
</tbody>
</table>

Defn1. An imperative machine is a pair $(A, c)$ where
• $A \subset (B^* \rightarrow B^*)$ is a finite set of actions (instructions)
• $c : B^* \rightarrow \mathbb{N}$ is a control function

Defn2. The halt instruction $halt \in A$ is the identity function on $B^*$. That is, $halt(b) = b$, for all $b \in B^*$. 
Model of computation

Defn 3. A sequential program $p$ of length $m$ is a sequence of instructions $p: \mathbb{N} \rightarrow A$ where $p(n) = \text{halt}$, for all $n \geq m$.

Defn 4. A thread is an execution of a sequential program $p$ on a machine $(A,c)$ which is defined by a sequence on $B^*$, $b_0, b_1, b_2, \ldots$, where $b_{n+1} = p(c(b_n))(b_n)$ for $n \geq 0$.

Defn 5. An of $(p_1 \parallel p_2)$, for sequential programs $p_1$ and $p_2$, is a sequence on $B^*$, $b_0, b_1, b_2, \ldots$, where $b_{n+1} = \text{choose} \{ p_1(c(b_n))(b_n), p_2(c(b_n))(b_n) \}$ for $n \geq 0$.

Essential problem

• For a sequential program (Defn 4):
  - Execution is determined by $b_0$ and $p$.
  - Two programs are considered equivalent if they compute the same partial function
    - halt on the same inputs and
    - return the same values for these inputs

• For concurrent threads (Defn 5):
  - How to determine equivalence of $(p_1 \parallel p_2)$ and $(p'_1 \parallel p'_2)$?
    - If all interleavings halt for the same initial state and give the same final state
    - Too many interleavings to reason about
  - Even when programs are equivalent when executed by Defn 4, they may no longer be equivalent when executed concurrently with other threads
  - Conclude: No useful notion of equivalence for threads

Essential problem

• Threads are wildly nondeterministic and job of the programmer is to prune away undesired nondeterminism.

• Argues that a better approach:
  - Nondeterminism should be explicitly added to programs only where needed

Example: observer pattern

Not thread safe:

```java
public void setValue(int nv) {
    value = nv;
    Iterator i = listeners.iterator();
    while (i.hasNext()) {
        ((Listener)i.next()).valueChanged(nv);
    }
}
```

```java
public void addListener(Listener nlis) {
    listeners.add(nlis);
}
```

```java
T1: Thread
T1: Thread
```

```java
ValueHolder
Value value
addListener(...) setValue(...) 
```

```java
Listener
listeners
valueChanged(...) 
```

Data Race!
Example: observer pattern

Listener

ValueHolder

\texttt{synchronized void setValue(int nv)}
\begin{verbatim}
  \texttt{value = nv;
  iterator i = listeners.iterator();
  while (i.hasNext()) {
    ((Listener)i.next()).valueChanged(nv);
  }
}\end{verbatim}

\texttt{synchronized void addListener(\ldots nil) \{ \}
\begin{verbatim}
  listeners.add(nlis);
}\end{verbatim}

Deadlock!

High-level race!

Aggressive pruning

\begin{itemize}
  \item Rigorous software engineering (testing, inspections-see anecdote)
  \item Impose total order on locks and acquire them in order
    \begin{itemize}
      \item inflexible and brittle
      \item non compositional
        \begin{itemize}
          \item locks needed are not specified in interface
          \item symmetric access becomes very difficult
        \end{itemize}
    \end{itemize}
  \item Design patterns for concurrent programming
    \begin{itemize}
      \item implementing a CP pattern is subtle and tricky
      \item properties of different design patterns do not compose
    \end{itemize}
  \item Some notable high level CP design patterns
    \begin{itemize}
      \item Transactions—work on a copy of the data, then \texttt{commit or abort}
        \begin{itemize}
          \item Good for intrinsically nondeterminate situations (multiple competing actors)
        \end{itemize}
      \item MapReduce—large scale distributed processing of huge data sets
        \begin{itemize}
          \item Encapsulate in libraries for non-experts to use
        \end{itemize}
    \end{itemize}
  \item Extend/constrain existing PL
    \begin{itemize}
      \item Cilk: extends C with new keywords \texttt{cilk, spawn, sync}
      \item Guava:
        \begin{itemize}
          \item distinguish read locks and write locks
          \item permit access by multiple threads to synchronized objects only
        \end{itemize}
      \item Promises (also called futures):
        \begin{itemize}
          \item rather than block, proceed with a proxy of data that another thread will provide
        \end{itemize}
      \item Szumo:
        \begin{itemize}
          \item specify resource needs in interface (synchronization contracts)
          \item permit access by thread to object only when the object is covered by contract
          \item middleware automates deadlock avoidance protocols to lock objects under contract on behalf of thread
        \end{itemize}
    \end{itemize}
  \item Formal program analysis to identify potential concurrency bugs
\end{itemize}
Coordination approach (Ptolemy II)

- Start with deterministic, composable mechanisms
- Introduce nondeterminism only where needed

Each icon is a Java program.
Rendezvous Director indicates CSP-like composition of components.
Merge indicates conditional rendezvous (2 possible 3-way rendezvous)

PN Director indicates components communicate via message passing with unbounded FIFO channels and blocking reads.
Annotation on Merge combinator indicates nondeterministic merge.

Advantages
- "...once you understand what the icons mean, the diagram very clearly expresses the observer pattern."
- "...everything about the diagram is deterministic except the explicitly nondeterministic interaction specified by the Merge block"
- can prove correctness properties
  - no deadlock
  - value consumer and observer see values in the same order

Challenges
- Chip away at homogeneity bias (language wars)
- Designing good coordination languages
- Scalability and modularity features
- Better computational model for concurrent computation