Banyan

A Framework for Distributing Tree-Structured Problems

Chris, David, Michelle
Context

• From class: MapReduce and Dryad
• Simple interfaces for distributed programming
But what about ...

```plaintext
fun divide_conquer (problem) {
  subproblems = divide(problem)
  subanswers =
    map divide_conquer subproblems
  combine subanswers
}

• It’s recursive!
```
Our contribution: Banyan

- Framework for distributing tree-structured recursive programs
- Begins computation on one processor; ships subproblems as they arise
- Uses Fly tuplespace for memoization
- Theorem prover case study
Outline

• Programmer interface
• Implementation
• Case study: theorem prover
• Results
• Future work and conclusion
Programmer interface

• Implement two primary functions:
  • workHere, to generate subproblems
  • childReturned, to combine answers

• Also
  • Generic message handler
  • Cooperative scheduling functions
  • Subproblem identifier for shared memory indexing
Architecture overview

Coordinator

Client

Worker

Worker

Worker

Shared Memory

Coordinator

Client

Worker

Worker

Worker

Shared Memory
Scheduling nodes

- Node priority expressed in tickets
- Job starts with fixed tickets; parents donate to children
- Active local nodes scheduled round-robin
- Subtrees shipped when local tickets exceed target
- Coordinator keeps workers balanced
Shared memory

• Fly tuple-space
  • Read, write, take
  • Read, take match on keyed template
• We use it to avoid duplicating work
  • Store node results
  • Check store on child create
  • Programmer supplies indexing scheme
Case study: Differential dynamic logic

• Interacting discrete and continuous components
• Hard parts:
  • Invariant generation
  • Quantifier elimination
DL as a Banyan Application

• Three concrete objects extending TreeNode
• Code snippet from AndNode:

```scala
def childReturned(child: Int, v: ReturnType): Unit =
  v match {
    case Proved(rl) =>
      numOpenChildren -= 1
      if(numOpenChildren <= 0) returnNode(Proved(rule))
    case GaveUp() =>
      returnNode(GaveUp())
  }
```

• It works!
Water tank example

- Proved
- Gave up
- Aborted
## Results – Water tank

<table>
<thead>
<tr>
<th>Workers</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2978</td>
</tr>
<tr>
<td>4</td>
<td>1213</td>
</tr>
</tbody>
</table>

2.4x speedup
Results – Bouncing ball

<table>
<thead>
<tr>
<th>Workers</th>
<th>T1 (s)</th>
<th>T2 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>209</td>
<td>147</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>62</td>
</tr>
</tbody>
</table>

3.2x speedup (avg)
Future work

• Optimize node distribution
• Pause/resume
• Dynamically add jobs, workers
• Fault tolerance
• Improve shared memory
  • Inexact matching
  • Multi-host flyspace
• Compare to simpler models
Conclusion

• We made a framework to facilitate distribution of tree-structured problems
• You can play with it:

www.cs.cmu.edu/~renshaw/banyan