Applying Dataflow and Transactions to Lee Routing

Chris Seaton, Daniel Goodman, Mikel Luján, and Ian Watson

University of Manchester

{seatonc,goodmand,mikel.lujan,watson}@cs.man.ac.uk

MULTIPROG 2012
23 January 2012
Paris
Aims

• Looking at general purpose programming on commodity systems

• Evaluate an implementation of dataflow combined with transactions

• Lee’s algorithm for circuit routing as a test application

• Impact on programmability

• Speedup
Multicore as commodity, need to parallelise irregular algorithms such as Lee

Dataflow

Transactions

Creation of parallelism

Access to shared state

Simple implementations that achieve speedup
Example application – circuit routing
Example application – circuit routing
Lee’s algorithm – sequential
Lee’s algorithm – sequential
Lee’s algorithm – sequential
Lee’s algorithm – parallel

- Lots of routes
- Find enough independent routes
- Where on the board will a route go?
- Very difficult to lock before starting
Dataflow

- Functional
- Functions scheduled when input ready
- Pass input from function to function
- All ready functions can be run in parallel
- Supports traditional parallelism - divide and conquer
Dataflow

innerProduct \((a, b) (c, d) = (a \times c) + (b \times d)\)
Dataflow

innerProduct \((a, b) \ (c, d) = (a \times c) + (b \times d)\)
Dataflow

innerProduct \((a, b) (c, d) = (a \times c) + (b \times d)\)
Transactional memory

- Semantic annotation of code that is to be executed atomically
- Often optimistic – roll back and retry
- Often implemented using locks, atomic instructions
- Suits irregular algorithms as dependencies can be handled only when they occur
Tools

- Scala
- Transactional memory: MUTS
- Dataflow: DFLib
- Teraflux project – [http://www.teraflux.eu](http://www.teraflux.eu)
Implementation of Lee

• Sequential
• Coarse locked
• Transactional: MUTS
• Dataflow + transactional: MUTS + DFLib
lock
  copy board state
unlock

... produce a solution ...

lock

is the solution still valid:
  save it to the board
else:
  retry it later

unlock

Accessing shared state – coarse lock
atomically:
copy board state

... produce a solution ...

atomically:
is the solution still valid:
save it to the board
else:
retry it later

Accessing shared state – transactions
for each core:
fork a new thread:
    loop while work:
        lock worklist
        take a route
        unlock
        ... solve the route ...
        lock solutions
        add to the list of solutions
        unlock

Scheduling – threads
solutions_thread = create collector thread (n)

for each route:
    route_thread = create thread (solveRoute)
    route_thread.arg1 ← board
    route_thread.arg2 ← route
    route_thread.arg3 ← boardState
    route_thread.output → solutions_thread

solutions_thread.output → …

Scheduling – dataflow
subset of routes currently scheduled on a hardware thread

Dataflow
Dataflow
Dataflow
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Lines of code</th>
<th>Parallel operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>251</td>
<td>0</td>
</tr>
<tr>
<td>Coarse lock</td>
<td>330 (+79)</td>
<td>11</td>
</tr>
<tr>
<td>Transactional</td>
<td>328 (+77)</td>
<td>6</td>
</tr>
<tr>
<td>Dataflow + transactional</td>
<td>300 (+49)</td>
<td>5</td>
</tr>
</tbody>
</table>

**Code metrics**
Experimental Design

We have simpler programs – do we still get a decent speedup?

- Commodity hardware
- Intel Core i7 920, 4 cores each with 2-way SMT
- SuSE 11.2, Linux 2.6
- Java 1.6, Scala 2.9, MUTS 1.1
- Wall clock run time, excluding setup and IO
- 10 repetitions with mean and SD recorded
Speedup relative to sequential
Conclusions

• Dataflow can be combined with transactions
• Lee shows certain properties that are currently difficult to parallelise
• Together dataflow and transactions are easier to program than on their own
• Together they produce performance similar to transactions on their own, and faster than coarse locks

http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS/
(or just search for “scala muts”)

Questions

Chris Seaton, Daniel Goodman, Mikel Luján, and Ian Watson

University of Manchester

{seatonc,goodmand,mikel.lujan,watson}@cs.man.ac.uk

Chris Seaton is an EPSRC funded student. Mikel Luján is a Royal Society University Research Fellow. The Teraflux project is funded by the European Commission Seventh Framework Programme.
val boardStateForFreeze = boardStateVar.take() // Lock
val privateBoardState = boardStateForFreeze.freeze
boardStateVar.put(boardStateForFreeze) // Unlock

...

val boardStateForLay = boardStateVar.take() // Lock

val verified = verifyRoute(route, solution, boardStateForLay)

if (verified)
    layRoute(route, solution, boardStateForLay)
else
    scheduleForRetry(route)

boardStateVar.put(boardStateForLay) // Unlock

Coarse lock
val privateBoardState = atomic { boardState.freeze }
...

atomic {
  if (verifyRoute(route, solution, boardState))
    layRoute(route, solution, boardState)
  else
    scheduleForRetry(route)
}

Transactional
Threads with a work list

val threads = for (n <- 0 until threadsCount) yield 
  new Thread(new Runnable() {
    def run() = {
      while (...) {
        var routes = routesVar.take()  // Lock
        
        routesVar.put(routes)  // Unlock

        if (route == null) {
          
        } else {
          val solution = solveRoute(board, route, boardStateVar)

          var solutions = solutionsVar.take()  // Lock
          solutions ::= solution
          solutionsVar.put(solutions)  // Unlock
        }
      }
    }
  })
val solutionCollector = 
    DFManager.createCollectorThread[Solution](routes.length)

for (route <- routes) {
    val routeSolver = DFManager.createThread(solveRoute _)
    routeSolver.arg1 = board
    routeSolver.arg2 = route
    routeSolver.arg3 = boardState
    routeSolver.arg4 = solutionCollector.token1
}

solutionCollector.addListener(solutionsOut)