Santa’s Groovy Parallel Helper

Jon Kerridge

NAPIER UNIVERSITY
EDINBURGH

Motivation

- Matt Pedersen’s and Jason Hurt’s submitted paper to CPA2008

Critique

- Their JCSP solution did not use two available synchronisation techniques
  - Bucket
    - A component into which one or more processes can **fallInto** thereby pre-empting themselves, becoming idle, until another process **flush**es all the processes thereby enabling their re-scheduling
  - Alting Barrier
    - A Barrier enables processes to synchronise such that the set of processes synchronising on the Barrier wait until they have all reached that point in their execution.
    - An Alting Barrier is one that can be used as part of a non-deterministic choice (Alternative)
    - Provides the CSP multi-way synchronisation primitive
- Further simplification by using Groovy Parallel Helper Classes

Bucket - methods

- **fallInto()**
  - The calling process is pre-empted
  - Process becomes idle consuming no processor resource and is associated with the Bucket
  - Any number of processes can fall into a bucket
- **flush()**
  - Must be called by a process that is never pre-empted in a Bucket
  - All the processes associated with the Bucket are rescheduled for execution
    - They may not execute immediately

Alting Barrier

- A possibly dynamic number of processes agree to synchronise on the Alting Barrier
- They do this either
  - absolutely by calling the AltingBarrier's **sync()** method
  - Process cannot withdraw from the synchronisation
- Or
  - They access the Alting Barrier by means of a guard in an Alternative (ALT)
- Only if the previously agreed number of processes have synchronised or are waiting on an ALT is the Alting Barrier selected as part of a non-deterministic choice

Reindeer Synchronisation

- Alting Barrier comprising
  - Santa Claus
  - Nine Reindeer
- Whenever Santa Claus and the nine reindeer have synchronised on the Alting Barrier
  - Given priority to deliver toys
  - Solely determined when all the reindeer synchronise because Santa checks for this possibility on each iteration
  - Minimal overhead is incurred by Santa
- Implemented as the stable Alting Barrier

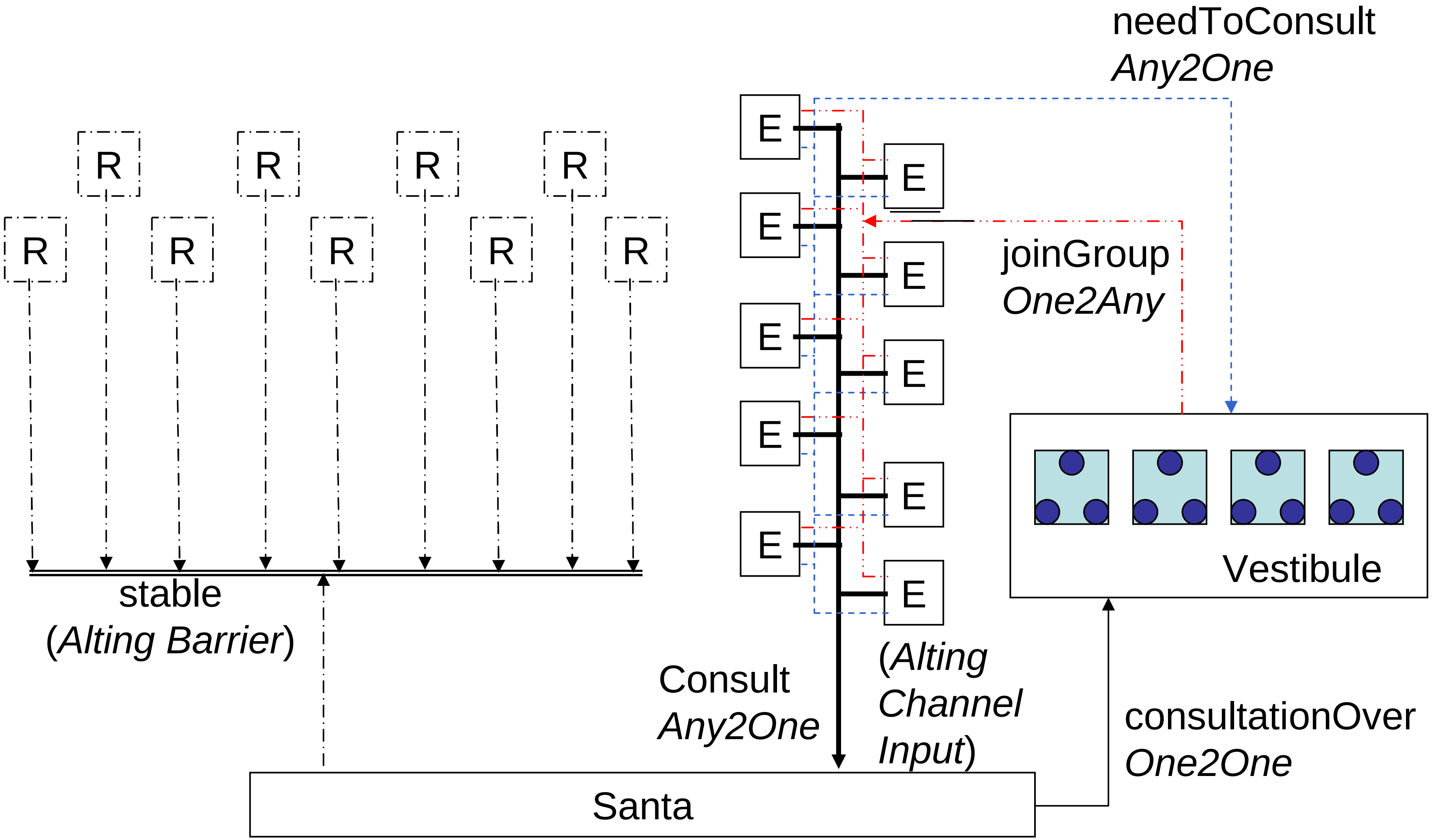
Vestibule

- Contains four groups, each implemented by a Bucket which can each hold up to three elves
- An Elf can tell the Vestibule they need to consult Santa
- The Vestibule then tells the Elf which group (Bucket) to join
- The Elf then **fallsInto()** the indicated Bucket
- The Elf then waits, idle in the Bucket until it is **flush**ed by the Vestibule.

Elf Synchronisation

- Whenever Santa finishes an elvin consultation he informs the Vestibule
  - The vestibule can then **flush()** the next group of elves, if any, so they can consult with Santa
- If Santa is idle and a third elf joins a group the Vestibule will **flush()** the group enabling them to consult with Santa Claus
  - Santa Claus does not have to check to see if there is a waiting group of elves

Architecture - Synchronisation



Reindeer

```
def AltingBarrier stable

while (true) {
        println "Reindeer ${number}: on holiday ... wish you were here, :)"
        timer.sleep (holidayTime + rng.nextInt(holidayTime))
        println "Reindeer ${number}: back from holiday ... ready for work, :("
        stable.sync()
        harness.write(number)
        harnessed.read()
        println "Reindeer ${number}: delivering toys . la-di-da-di-da-di-da, :)"
        returned.read()
        println "Reindeer ${number}: all toys delivered ... want a holiday, :("
        unharness.read()
}
```

Elf

```
while (true){
        println "Elf ${number}: working, :)"
        timer.sleep ( workingTime + rng.nextInt(workingTime))
        needToConsult.write(1)
        def group = joinGroup.read()
        groups[group].fallInto()

        // idle in Bucket awaiting flush()

        consult.write(number)
        println "Elf ${number}: need to consult Santa, :("
        consulting.read()
        println "Elf ${number}: about these toys ... ???"
        negotiating.write(1)
        consulted.read()
        println "Elf ${number}: OK ... we will build it, bye, :("
}
```

Consult Channel - Elves to Santa

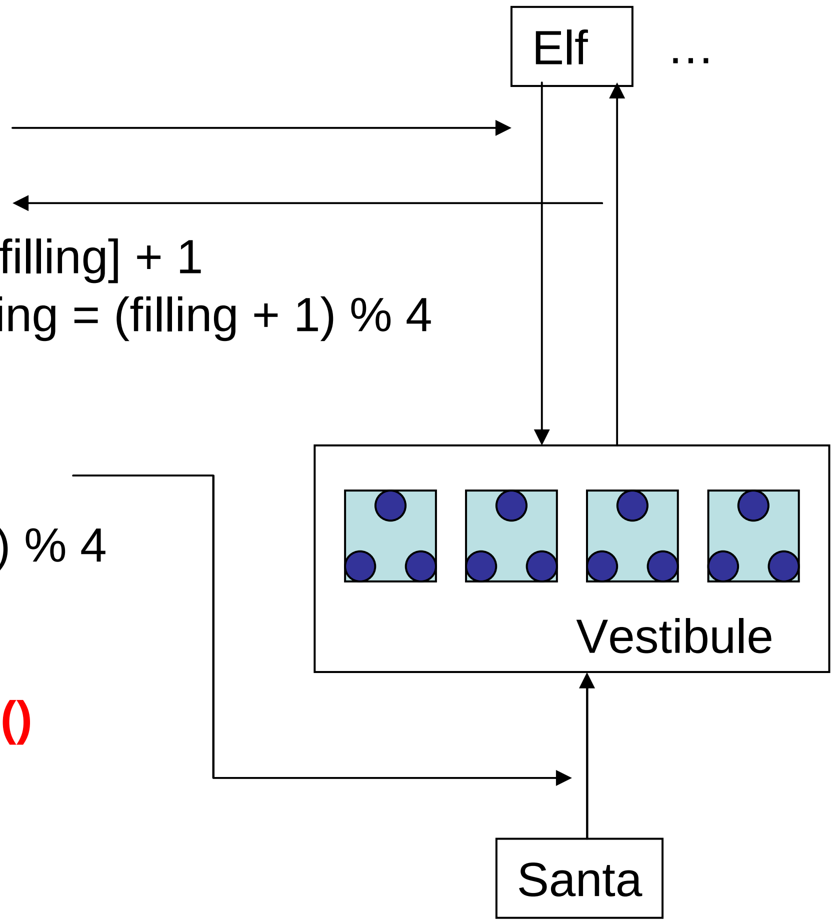
- Any to One
  - Each elf can write to Santa
- However
  - At any one time only three elves are flushed
  - Hence Santa can expect exactly three communications
  - It does not matter which elf communicates first
  - Provided the other two elf communications are read
- Similarly for the Vestibule channel communications
  - needToConsult (Any2One)
  - joinGroup (One2Any)

Vestibule – Set Up

```
def flush = new Skip()
def vAlt = new ALT ([needToConsult, consultationOver, flush])
def int index = -1
def int filling = 0
def int removing = 0
def counter = [0, 0, 0, 0]
def NEED = 0
def OVER = 1
def FLUSH = 2
def preCon = new boolean[3]
preCon[NEED] = true
preCon[OVER] = true
preCon[FLUSH] = false
openForBusiness.read()
```

Vestibule – Main Loop

```
while (true){
        index = vAlt.select(preCon)
        switch (index) {
        case NEED:
                needToConsult.read()
                joinGroup.write(filling)
                counter [filling] = counter [filling] + 1
                if (counter [filling] == 3) filling = (filling + 1) % 4
                break
        case OVER:
                consultationOver.read()
                removing = (removing + 1) % 4
                break
        case FLUSH:
                groups [removing].flush()
                counter [removing] = 0
                break
        }
        preCon [FLUSH] = ( counter [removing] == 3 )
}
```



Santa – Set Up

```
def AltingBarrier stable
def ChannelInput consult

    def REINDEER = 0
    def ELVES = 1
    def rng = new Random()
    def timer = new CSTimer()

    def santaAlt = new ALT([stable, consult])
    openForBusiness.write(1)
```

Santa – Reindeer Choice

```
while (true) {
        index = santaAlt.priSelect()
        switch (index) {
        case REINDEER :
                def id = []
                println "Santa: ho-ho-ho ... the reindeer are back"
                for ( i in 0 .. 8){
                        id[i] = harness.read()
                        println "Santa: harnessing reindeer ${id[i]} ..."
                }
                println "Santa: mush mush ..."
                for ( i in 0 .. 8) harnessed.write(1)
                timer.sleep ( deliveryTime + rng.nextInt(deliveryTime))
                println "Santa: woah ... we are back home"
                for ( i in 0 .. 8) returned.write(1)
                for ( i in 0 .. 8) {
                        println "Santa: unharnessing reindeer ${id[i]}"
                        unharnessList[id[i]].write(1)
                }
                break
```

Santa – Elf Choice

```
case ELVES:
        def id = []
        id[0] = consult.read()
        for ( i in 1 .. 2) id[i] = consult.read()    // expecting precisely 2 more reads
        println "Santa: ho-ho-ho ... some elves are here!"
        for ( i in 0 .. 2){
                consulting[id[i]].write(1)
                println "Santa: hello elf ${id[i]} ..."
        }
        for ( i in 0 .. 2) negotiating.read()
        println "Santa: consulting with elves ..."
        timer.sleep ( consultationTime + rng.nextInt(consultationTime))
        println "Santa: OK, all done - thanks!"
        for ( i in 0 .. 2){
                consulted[id[i]].write(1)
                println "Santa: goodbye elf ${id[i]} ..."
        }
        consultationOver.write(1)
        break
```

Result

- Shared Memory (Thread based models)
  - C# - 642 lines
  - C - 420 lines
  - Java - 564 lines
  - Groovy - 322 lines
- Distributed Memory
  - MPI - 352 lines
- Process Oriented
  - JCSP - 315 lines
- Groovy Parallel – 215 lines
  - 32% reduction over JCSP !!!

Conclusion for Management at the North Pole

Santa

Should Use

Groovy

Parallel !!