Workbook

Zoe Wall
40182161@napier.ac.uk
Edinburgh Napier University - Fundamentals of Parallel Systems (SET09109)

Exercise 2

2.1 Multiplier Process

Code

Listing 1: "Multiplier.groovy"

```
1   void run()
2   {
3       def i = inChannel.read()
4       while (i > 0) {
5           // write i * factor to outChannel
6           outChannel.write(i*factor)
7           // read in the next value of i
8           i = inChannel.read()
9       }
10      outChannel.write(i)
11  }
```

Listing 2: "Consumer.groovy"

```
1   while ( i > 0 )
2   {
3       //insert a modified println statement
4       println "The output is : ${i}"
5       i = inChannel.read()
6   }
```

Listing 3: "RunMultiplier.groovy"

```
1   def processList = [ new Producer ( outChannel: connect1.out() ),
2
3       //insert here an instance of multiplier with a multiplication factor of 4
4       new Multiplier ( inChannel: connect1.in(),
5                        outChannel: connect2.out(),
6                        factor: 4),
7       new Consumer ( inChannel: connect2.in() )
8   ]
```

Output

```
next: 1
next: The output is : 4
4
next: The output is : 16
10
next: The output is : 40
0
Finished
```

Figure 1: **Exercise 2-1** - Output from Run Multiplier program.

Explanation The *Multiplier.groovy* process (see Listing 1) is inserted into the process list between the producer and consumer. The *Producer.groovy* process outputs an integer, that has been provided, to the Multiplier process which then outputs each integer multiplied by a constant factor which is set in the constructor of the Multiplier instance (see line 6 in Listing 3). The *Consumer.groovy* process prints a meaningful output to the console.

2.2 Integer Sets

Code

Listing 4: "ListToStream.groovy"

```
1  while (inList[0] != -1)
2  {
3      // hint: output list elements as single integers
4      for ( i in 0 ..< inList.size)outChannel.write(inList[i])
5          inList = inChannel.read()
6  }
```

Listing 5: "CreateSetsOfEight.groovy"

```
1  while (v != -1)
2  {
3      for ( i in 0 .. 7 )
4      {
5          // put v into outList and read next input
6          outList[i] = v
7          v = inChannel.read()
8      }
9      println " Eight Object is ${outList}"
10 }
```

Listing 6: "GenerateSetsOfThree.groovy"

```
1  void run()
2  {
3      def threeList = [
4          [1, 2, 3],
5          [4, 5, 6],
6          [7, 8, 9],
7          [10, 11, 12],
8          [13, 14, 15],
9          [16, 17, 18],
10         [19, 20, 21],
11         [22, 23, 24],
12         [-1, -1, -1]]       // terminating list
13     for ( i in 0 ..< threeList.size)outChannel.write(threeList[i])
14     //write the terminating List as per exercise definition
15 }
```

Output

```
 Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]
 Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]
 Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]
Finished
```

Figure 2: **Exercise 2-2** - Output from Run Three to Eight program.

Exercise Questions

What change is required to output objects containing six integers?

Within the *CreateSetsOfEight.groovy* process, change the number of iterations of the for loop. See line 3 of listing 5. The new line should read

Listing 7: "CreateSetsOfEight.groovy - Change required to output objects containing six integers"

```
1  //for (i in 0 .. 7) {
2  for (i in 0 .. 5) {
```

which will now create lists containing 6 integers. This can be improved by changing the number to a variable which is set through the constructor of the process.

How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12)?

The process can be improved by creating a parameter for the size of the output list, bearing in mind the iterations start from zero so the required size should be minus one of the input parameter. This can either be a variable that is set within the constructor of the *CreateSetsOfEight.groovy* process or by user input from the console.

What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7) ?

Some integers from the input stream will not be outputted to the console as they do not make up a full set of numbers. So the set cannot be filled and therefore the set will not display as the process cannot finish.

Exercise 3

3.1 Reversing GIntegrate

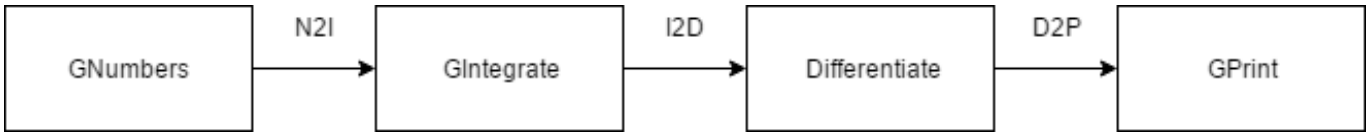

Figure 3: **Exercise 3-1** - Network for process reversing the effect of *GIntergrate.groovy*.

GNumbers.groovy outputs a stream of integers starting at zero and incrementing by one each time. *GIntegrate.groovy* increments the stream by an increasing number each time so the difference between the output is increasing. To negate this effect the output from the network should be equal to the output from the initial GNumbers process.

Minus Process

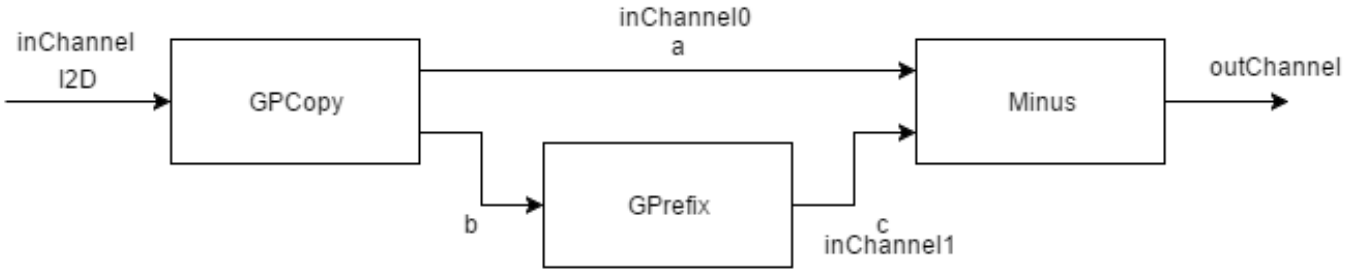

Figure 4: **Exercise 3-1** - Network showing processes for *Differentiate.groovy*.

Explanation The minus process works by copying the output value from *GIntegrate.groovy*, e.g 0, 1, 3, 6. One copy is sent straight to inChannel0 of the *Minus.groovy* process, and the other is sent to *GPrefix.groovy* which outputs the stream with a leading zero into inChannel1 - see Figure 4. The *Minus.groovy* process reads both inputs in parallel and minuses the second from the first resulting in an output stream of incrementing numbers (Figure 5).

Code

Listing 8: "Minus.groovy"

```
1  ProcessRead read0 = new ProcessRead ( inChannel0)
2  ProcessRead read1 = new ProcessRead ( inChannel1)
3  def parRead2 = new PAR ( [ read0, read1 ] )
4
5  while (true) {
```

```
6       parRead2.run()
7       // output one value subtracted from the other
8       // be certain you know which way round you are doing the subtraction!!
9       outChannel.write(read0.value - read1.value)
10  }
```

Listing 9: "Differentiate.groovy - see Figure 4 for network of this list"

```
1   def differentiateList = [ new GPrefix ( prefixValue: 0,
2                                 inChannel: b.in(),
3                                 outChannel: c.out() ),
4                             new GPCopy ( inChannel: inChannel,
5                                 outChannel0: a.out(),
6                                 outChannel1: b.out() ),
7                             // insert a constructor for Minus
8                             new Minus (inChannel0: a.in(),
9                                 inChannel1: c.in(),
10                                outChannel: outChannel)
11                            ]
```

Output

```
Differentiated Numbers
0
1
2
3
4
5
6
7
8
9
10
```

Figure 5: **Exercise 3-1** - Output from the Differentiate Process.

Negator Process

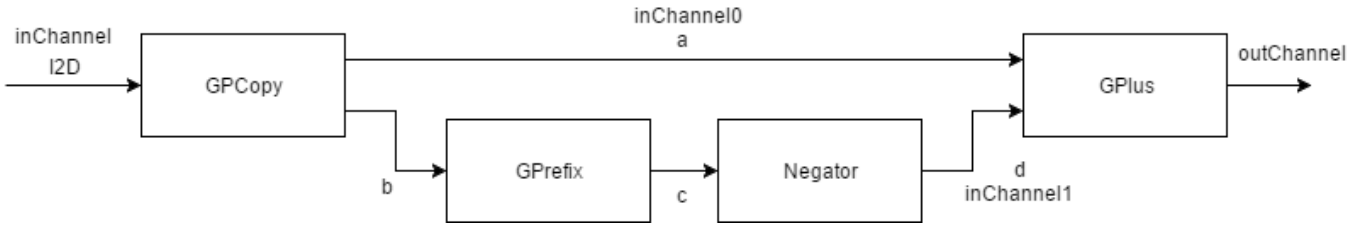

Figure 6: **Exercise 3-1** - Network showing processes for *DifferentiateNeg.groovy* (replacement for Differentiate process within Figure 3)

Explanation

Code

Listing 10: "DifferentiateNeg.groovy"

```
1   def differentiateList = [ new GPrefix ( prefixValue: 0,
2                                 inChannel: b.in(),
3                                 outChannel: c.out() ),
4                             new GPCopy ( inChannel: inChannel,
5                                 outChannel0: a.out(),
6                                 outChannel1: b.out() ),
7                             //insert a constructor for Negator
8                             new Negator ( inChannel: c.in(), outChannel: d.out()),
9                             new GPlus  ( inChannel0: a.in(),
10                                inChannel1: d.in(),
11                                outChannel: outChannel )
12                            ]
```

Output

Exercise Questions

Which is the more pleasing solution and why? In this case, even though the Negator solution requires an extra process, it can be argued that this is more pleasing as there is less room for error introduced by the order of the values within the Minus solution. However, a more appropriate solution to negating the effects of GIntegrate would be not to send the values through it in the first place.

3.2 Exercise 3-2

3.3 Exercise 3-3

Exercise 4-1

When the read line is removed the output alternates between the incrementing reset value and the original numbers. This is because the original value is not removed from the system if the channel is not read from.

5.1 Varying the delay times makes no difference to the process output, it works as expected regardless of changing the times. This happens because it uses preconditions to

the latter situatuon is more

The latter is the more elegant solution because it avoids nested loops.

References

[1] J. Malkevitch, “Sales and chips,” *Accessed: October 2016*. www.ams.org.

[2] M. Freiberger, “The travelling salesman,” *Accessed: November 2016*. www.plus.maths.org.

[3] D. Johnson and L. McGeoch, “The travelling salesman problem: A case study in local optimization,” pp. 7–8, 1995.

[4] C. Nilsson, “Heuristics for the travelling salesman problem,” pp. 1–3, 2003.

4 Appendix