Let's Do It in Parallel:

Groovy Parallel Programming for Java Developers

Jon Kerridge

School of Computing

Edinburgh Napier University, Scotland

Preface

The aim of this book is to show both students and practitioners that concurrent and parallel programming does not need to be as hard as it is often portrayed and in fact is often easier that building the equivalent sequential system. This will be achieved by presenting a set of example systems that demonstrate the underlying principles of parallel system design based upon real world examples. Each chapter will discuss the complete implementation of such a system, rather than presenting fragments of solutions. The approach will therefore be founded in principled engineering rather than a detailed exploration of the scientific underpinning. The science has been explored in many books but these have not demonstrated the engineering aspects of actually designing and building parallel systems.

For the purposes of this book; Concurrent means a system built from a set of processes that execute on a single processor. Parallel means that more than one processor is used to execute the processes and these communicate over some form of network. Within a parallel system it is likely that some of the processors will run some processes concurrently.

The book will use as its underpinning parallel environment a package called JCSP (Communicating Sequential Processes for Java) that is available under the LGPL software licence from the University of Kent, Canterbury UK. This package implements the Communicating Sequential Process concepts developed by Professor Hoare some 25 years ago in a form that makes them easily accessible to the programmer. The book's emphasis is on the engineering of parallel systems using these well-defined concepts without delving into their detailed theoretical aspects. The JCSP package essentially hides Java's underlying thread model from the programmer in a manner that allows easy implementation of concurrent and parallel systems. It is immaterial whether a process is executed concurrently or in parallel, the process definition remains the same.

Understanding the principles behind parallel processing is an increasingly important skill with the advent of multi-core processors. Much effort has been made by processor manufacturers to hide the underlying parallel design techniques by providing tools that will take an existing code and extract some parallelism from it. This hides the real need to actually design and build parallel systems from the outset. Far too many people have been put off parallel processing because they believe that they have to understand the underlying thread model supplied as part of the language or operating system environment. The goal of the book is to dispel all these misconceptions and show that parallel system can be built quite easily with a very few simple design patterns and that such parallel systems can be easily implemented on a single processor or a collection of networked processors. Furthermore the advent of multi-core processors means that we can now start to build genuinely parallel systems for the commonest desktop workstations in which we can exploit the inherent parallelism more easily once we have the tools required to place a process on a specific core. The extension to a network of multi-core processors becomes even easier. Equally important is that the same design principles can be used to build mobile systems that permit interactions between mobile devices and fixed services using wireless and Bluetooth technology.

Background

The book results from a module taught during the spring semester to master's students, though the approach would be applicable to senior undergraduates and professional

programmers. As part of the module, students were asked to complete a practical portfolio and are included in the book. The source coding for all the examples and for solutions to the practical problems is also available. A set of PowerPoint slides is also available for instructors.

Why Java and Groovy and Eclipse?

Java is widely used and, increasingly in mobile applications, where the ability to build parallel systems is crucial. Most user interfaces are in fact concurrent in nature but you would not think so given the contortions a Java programmer has to go to make them appear sequential. Groovy is a scripting language for the Java platform, currently being developed and progressed through the Sun Java release programme. The addition of a small number of helper classes has made the underlying parallel design concepts far more easily accessible as the amount of code that has to be written to create a working application is dramatically reduced.

Eclipse is becoming the de facto standard for Integrated Software Development Environments and as such has the appropriate plug-ins for Java and Groovy which have been extended to include the Groovy Parallel helper classes.

It is assumed that the reader of this book has some familiarity with Java and Java-like languages and sufficient understanding of object-oriented principles to cope with data encapsulation and the concept of `get()` and `set()` methods. An awareness of the `java.awt` package will be of benefit when user interfaces are discussed.

Parallelism, as such, has been a topic that has been avoided by many educational establishments, possibly because of the lack of appropriate software environments and approachable tools. It also suffers from the underlying models being overly complex and difficult to reason about. This book addresses these issues by providing a model of parallel programming based in the Object Oriented paradigm that builds upon many year's of research into the design and implementation of parallel systems. It attempts to demonstrate that parallel programming is not hard and perhaps should be considered as a more appropriate first design environment for the teaching of programming as it is closer to the way in which humans understand and investigate solutions to problems.

Organisation of the Book

To be done when finally organised

Acknowledgements

Lots but in due course

Contents

1 Basic Concepts

The fundamental concepts that we shall be dealing with are:

1. processes
2. channels,
3. timers
4. alternatives

In comparison to other *concurrent* and *parallel* based approaches, the list is very small but that is because we are dealing with higher-level concepts and abstractions. Therefore it is much easier for the programmer to both build parallel systems and also to reason about their behaviour. One of the key aspects of this style of parallel system design is that processes can be composed into larger networks of processes with a predictable overall behaviour.

1.1 Process

A process, in its simplest form, defines a sequence of instructions that are to be carried out. Typically, a process will communicate with another process using a channel to transfer data from one to the other. In this way a network of processes collectively provide a solution to some problem. Processes have only one method, `run()`, which is used to invoke the process. A list of process instances is passed to an instance of a `PAR` object which, when `run`, causes the parallel execution of all the processes in the list. A `PAR` object only terminates when all the processes in the list have themselves terminated.

A process encapsulates the data upon which it operates. Such data can only be communicated to or from another process and hence all data is private to a process. Although a process definition is contained within a `class` definition, there are no explicit methods defined by which any property or attribute of the process can be accessed.

A network of processes can be invoked concurrently on the same processor, in which case the processor is said to interleave the operations of the processes. The processor can actually only execute one process at a time and thus the processor resource is shared amongst the concurrent processes.

A network of processes can be created that runs on many processors connected by some form of communication mechanism, such as a TCP/IP based network. In this case the processes on the different processors can genuinely execute at the same time and thus are said to run in parallel. In this case some of the processors may invoke more than one process and so an individual processor may have some processes running concurrently but the complete system is running in parallel. The definition of a process

remains the same regardless of whether it is executed concurrently or in parallel. Furthermore the designer does not have to be aware, when the process is defined, whether it will execute concurrently or in parallel.

A network of proceses can be run in parallel on a multi-core processor in such a way that the processes are executed on different cores. We can thus exploit multi-core processors directly by the use of a process based programming environment. The exploitation of multi-core processors will result in those processes running on the same core executing concurrently and thos on different cores in parallel. Currently, the ability to control the placement of specific processes on specific cores is limited by the underlying Java environment.

Throughout the rest of this book we shall refer to a network of parallel processes without specifically stating whether the system is running concurrently or in parallel. Only when absolutely necessary will this be differentiated.

1.2 Channel

A channel is the means by which a process communicates with another process. A channel is a one-way, point-to-point connection between two processes. One process `write`s to the channel and the other `read`s from the channel. Channels are unbuffered and are used to transfer data from the outputting (writing) process to the inputting (reading) process. If we need to pass data between two processes in both directions then we have to supply two channels, one in each direction. Channels synchronise the processes to pass data from one to the other. Whichever process attempts to communicate first waits, idle, using no processor resource until the other process is ready to communicate. The second process attempting to communicate will discover this situation, undertake the data transfer and then both processes will continue in parallel, or concurrently if they were both executed on the same processor. It does not matter whether the inputting or outputting process attempts to communicate first the behaviour is symmetrical. At no point in a channel communication interaction does one process cycle round a loop determining whether the other process is ready to communicate. The implementation uses neither *polling* nor *busy-wait-loops* and thus does not incur any overhead.

This describes the fundamental channel communication mechanism; however, within the parallel environment it is possible to create channels that have many reader and / or writer processes connected to them. In this case the semantics are a little more complex but in the final analysis the communication behaves as if it were a one-to-one communication.

When passing data between processes over a channel some care is needed because, in the Groovy environment, this will be achieved by passing an object reference if both processes are executing concurrently on the same processor. In order that neither of the processes can interfere with the behaviour of each other we have to ensure that a process does not modify an object once it has been communicated. This can be most easily achieved by always defining a new instance of the object which the receiving process can safely modify.

If the communication is between processes on different processors this requirement is relaxed because the underlying system has to make a copy of the data object in any case. An object reference has no validity when sent to another processor. Such a data object has to implement the `Serializable` interface.

In the processes are running on a multi-core processor then they should be treated as processes running concurrently on the same processor because such processes can share the same caches and thus processes will be able to access the same object reference.

1.3 Timers

A key aspect of the real world is that many systems rely on some aspect of time, either absolute or relative. Timers are a fundamental component of a parallel programming environment together with a set of operations. Time is derived from the processor's system clock and has millisecond accuracy. Operations permit the time to be read as an absolute value. For example, processes can be made to go idle for some defined, sleep, period. Alarms can be set, for some future time, and detected so that actions can be scheduled. A process that calls the `sleep()` method or is waiting for an alarm is idle and consumes no processor resource until it resumes execution.

1.4 Alternatives

The real world in which we interact is *non-deterministic*, which means that the specific ordering of external events and communications cannot be predefined in all cases. The programming environment therefore has to reflect this situation and permit the programmer to capture such behaviour. The *alternative* captures this behaviour and permits selection between one or more input communications, timer alarms and other synchronisation capabilities. The events over which the alternative makes its selection are referred to as guards. If one of the guards is ready then that one is chosen and its associated process carried out. If none of the guards are ready then the alternative waits, doing nothing, consuming no processor resource until one is ready. If more than one is ready, it chooses one of the ready guards according to some selection criterion. The ability to *select* a ready guard is a crucial requirement of any parallel programming environment that is going to model the non-deterministic real world.

1.5 Summary

This brief chapter has defined the terms we are going to use during the rest of the book. From these basic concepts we are going to build many example parallel systems simply by constructing networks of processes, connected by channels, each contributing, in part, to the solution of a problem. Whether the network of processes is run in parallel over a network, in a multi-core processor, or concurrently on a single processor has no bearing upon the design of the system. In some systems the use of multiple processors may be determined by the nature of the external system and environment to which the computer system is connected.

2 Producer Consumer: A Fundamental Design Pattern

For many people, the first program they write in a new language is to print "Hello World", followed by the inputting of a person's name so the program can be extended to print "Hello *name*". In the parallel world this is modified to a program that captures one of the fundamental design patterns of parallel systems, namely a Producer – Consumer system.

A Producer process is one that outputs a sequence of distinct data values. A Consumer process is one that inputs a stream of such data values and then processes them in some way. The composition of a Producer and Consumer together immediately generate some interesting possibilities for things to go wrong. What happens if?

5. The Producer is ready to output some data before the Consumer is ready

6. The Consumer is ready to input but no data is available from the Producer

In many situations, the programmer would resort to introducing some form of buffer between the Producer and Consumer to take account of any variation in the execution rate of the processes. This then introduces further difficulties in our ability to reason about the operation of the combined system; such as the buffer becomes full so the Producer has to stop outputting, or conversely it becomes empty and the Consumer cannot input any data. We have just put off the decision. In fact, we have made it much harder to both program the system and to reason about it. In addition, we now have to consider the situation when the buffer fails for some reason. Fortunately, the definitions of process and channel given in Chapter 1 come to our rescue.

If the Producer process is connected to the Consumer process by a channel then we know that the processes synchronise with each other when they transfer data over the channel. Thus if the Producer tries to output data before the Consumer is ready to input then the Producer waits until the Consumer is ready and vice-versa. It is therefore impossible for any data to be lost or spurious values created during the data communication.

2.1 A Parallel Hello World

2.1.1 Hello World Producer

The producer process for Hello-World is shown in Listing 2-1. Line {1}[1] specifies the imported package for JCSP. Line {2-10} defines the class `ProduceHW` that implements the interface `CSProcess`, which

[1] The notation { } in the text indicates the line number, or numbers, or range of lines in a Listing. All lines in a chapter are unique.

defines a single method `run()` that is used to invoke the process. The interface `CSProcess` is contained in the package `org.jcsp.lang`.

```
01    import org.jcsp.lang.*

02    class ProduceHW implements CSProcess {

03      def ChannelOutput outChannel

04      void run() {
05        def hi = "Hello"
06        def thing = "World"
07        outChannel.write ( hi )
08        outChannel.write ( thing )
09      }
10    }
```

Listing 2-1 Hello World Producer Process

The only class property, `outChannel` {3}, of type `ChannelOutput`, is the channel upon which the process will output using a `write` method. Strictly, Groovy does not require the typing of properties, or any other defined variable, however, for documentation purposes we adopt the convention of giving the type of all properties. This also has the advantage of allowing the compiler to check type rules and provides additional safety when processes are formed into process networks.. Each process has only one method, the `run()` method as shown starting at line {4}. Two variables are defined {5, 6} that hold the strings "Hello" and "World" respectively. These are then written in sequence to `outChannel` {7, 8}.

2.1.2 Hello World Consumer

The `ConsumeHW` process, Listing 2-2, has a property `inChannel` {13} of type `ChannelInput`. Such channels can only read objects from the channel using a `read` method. Its `run()` method firstly, reads in two variables from its `inChannel` {15, 16}, which are then printed {17} to the console window with preceding and following new lines (\n). The notation `${v}` indicates that the variable `v` should be evaluated to its `String` representation

```
11    import org.jcsp.lang.*

12    class ConsumeHW implements CSProcess {

13      def ChannelInput inChannel

14      void run() {
15        def first = inChannel.read()
16        def second = inChannel.read()
17        println "\n${first} ${second}!\n"
18      }
19    }
```

Listing 2-2 Hello World Consumer Process

2.1.3 Hello World Script

Figure 2-1 shows the process network diagram for this simple system comprising two processes `ProduceHW` and `ConsumeHW` that communicate over a channel names `connect`.

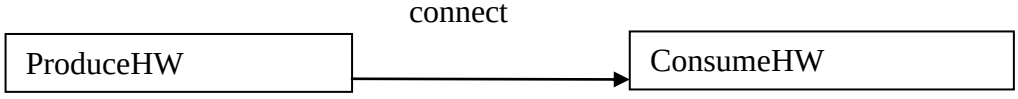

Figure 2-1 Producer Consumer Process Network

The script for executing the processes `ProduceHW` and `ConsumeHW` is shown in Listing 2-3.

```
20     import org.jcsp.lang.*
21     import org.jcsp.groovy.PAR

22     One2OneChannel connect = Channel.createOne2One()

23     def processList = [
24                           new ProduceHW ( outChannel: connect.out() ),
25                           new ConsumeHW ( inChannel: connect.in() )
26                         ]

27     new PAR (processList).run()
```

Listing 2-3 Hello World Script

The package `org.jcsp.groovy` {21} contains the definitions of the Groovy parallel helper classes. See Appendix 2 for a more complete specification and description of their use. The `PAR` class {27} causes the parallel invocation of a list of processes. This is achieved by calling the `run()` method of `PAR`, which in turn causes the execution of the `run()` method of each of the processes in the list.

The channel `connect` is an interface of type `One2OneChannel` {22}. The channel is created by means of a static method `createOne2One` in the class `Channel` contained within the package `org.jcsp.lang`. The `processList` {23} comprises an instance of `ProducerHW` with its `outChannel` property set to the `out` end of `connect`, and the processes `ConsumerHW` with its `inChannel` property set to the `in` end of `connect` {24, 25}.

The underlying JCSP[2] attempts, as far as possible, to ensure networks of processes are connected in a manner that can be easily checked. The channel `connect` {22} is defined as a `One2OneChannel` interface and therefore it has one output end and one input end. These are defined by the methods `out()` {24} and `in()` {25} respectively. A class that contains a property of type `ChannelOutput` must be passed an actual parameter that has been defined with a call to `out()` and within that process only `write()` method calls are permitted on the channel property. The converse is true for input channels. In all process network diagrams the arrow head associated with a channel will refer to the input end of the channel.

The output from executing this script is shown in Output 2-1.

```
Hello World!
```

Output 2-1 Output from Hello World Script

2.2 Hello Name

The Hello Name system is a simple extension of the Hello World program. The only change is that the `ProducerHN` {30} process asks the user for their name and then sends this to the `Consumer` process as the `thing` variable {34} in Listing 2-4.

```
28     import phw.util.*
29     import org.jcsp.lang.*

30     class ProduceHN implements CSProcess {

31       def ChannelOutput outChannel

32       void run() {
33         def hi = "Hello"
34         def thing = Ask.string ("\nName ? ")
35         outChannel.write ( hi )
36         outChannel.write ( thing )
37       }
38     }
```

Listing 2-4 The ProduceHN Process

[2] The JCSP or Communicating Sequential Processes for Java comprises a set of packages that implement various aspects of support for parallelism using CSP concepts for the Java Virtual Machine.

The package `phw.util` contains some simple console interaction methods that can be used to obtain input from the user from the console window, see Appendix 3. The `Ask.string` method outputs the prompt "`Name ?`" after a new line and the user response is then placed into the variable `thing` {34}.

The Consumer process remains unaltered from the version shown in Listing 2-2 except that the name of the process is changed to `ConsumeHN`. Similarly, the script to run the processes is the same as Lisitng 2-3 except that the names of the processes have been changed to `ProduceHN` and `ConsumeHN`. A typical output from the execution of the script is shown in Output 2-2, where user typed input is shown in *italics*.

```
Name ? Jon

Hello Jon!
```

Output 2-2 Output from Hello Name Network

2.3 Processing Simple Streams of Data

The final example in this chapter requires the user to type in a stream of integers into a producer process that writes them to a consumer process, which then prints them. The specification of the `Producer` process is given in Listing 2-5.

```
39    import phw.util.*
40    import org.jcsp.lang.*

41    class Producer implements CSProcess {

42      def ChannelOutput outChannel

43      void run() {
44        def i = 1000
45        while ( i > 0 ) {
46          i = Ask.Int ("next: ", -100, 100)
47          outChannel.write (i)
48        }
49      }
50    }
```

Listing 2-5 The Producer Process

The `run()` {43} method is formulated as a `while` loop {45-48}, which is terminated as soon as the user inputs zero. The input integer value is obtained using the `Ask.Int` method that will ensure that any input lies between -100 to 100 {46}. The `while` loop has been structured to ensure the final zero is also output to the `Consumer` process.

```
51    import org.jcsp.lang.*

52    class Consumer implements CSProcess {

53      def ChannelInput inChannel

54      void run() {
55        def i = 1000
56        while ( i > 0 ) {
57          i = inChannel.read()
58          println "the input was : ${i}"
59        }
60        println "Finished"
61      }
62    }
```

Listing 2-6 The Consumer Process

The Consumer process is shown in Listing 2-6. The `Consumer` {52} process reads data {57} from its input channel, `inChannel` {53}, which is then printed {58}. Once a zero is read the `while` loop {56-59} terminates resulting in the printing of the "`Finished`" message {60}.

The script that causes the execution of the network of processes is shown in Listing 2-7 which is very similar to the previous script shown in Listing 2-3, the only change being, the names of the processes that make up the list of processes {67, 68}.

```
63     import org.jcsp.lang.*
64     import org.jcsp.groovy.*

65     One2OneChannel connect = Channel.createOne2One()

66     def processList = [
67                         new Producer ( outChannel: connect.out() ),
68                         new Consumer ( inChannel: connect.in() )
69                       ]

70     new PAR (processList).run()
```

Listing 2-7 The Producer Consumer System Script

Output from a typical execution of the processes is given in Output 2-3.

```
next: 1
next: the input was : 1
2
the input was : 2
next: 3
the input was : 3
next: 0
the input was : 0
Finished
```

Output 2-3 Typical Results from Producer Consumer System

The output, especially when executed from within Eclipse, can be seen to be correct but the output is somewhat confused as we have two processes writing concurrently to a single console at the same time; both processes use println statements {46 (implicitly), 58, 60}. We therefore have no control of the order in which outputs appear and these often become interleaved. A process called GConsole is available in the package org.jcsp.groovy.plugAndPlay that creates a console window with both an input and output area. This process can be used to provide a specific console facility for a given process. Many such GConsole processes can be executed in a network of processes as required. Its use will be demonstrated in later chapters.

2.4 Summary

This chapter has introduced the basic use of many simple JCSP concepts. A set of simple Producer - Consumer based systems have been implemented and output from these systems has also been given. These basic building blocks of processes and channels, with their simple semantics are the basis for all the parallel systems we shall be building throughout the rest of this book.

2.5 Exercises

7. Using Listing 2-7 as a basic design implement and test a new process called `Multiply` that is inserted into the network between the `Producer` and `Consumer` processes which takes an input value from the `Producer` process and multiplies it by some constant `factor` before outputting it to the `Consumer` process. The multiplication `factor` should be one of the properties of the `Multiply` process. To make the output more meaningful you may want to change the output text in the `Consumer` process. You should reuse the `Producer` process from the `ChapterExamples` project in `src` package `c2`.

8. A system inputs data objects that contain three integers. It is required to output the data in objects that contain eight integers. Write and test a system that undertakes this operation. The process `ChapterExercises/src/c2.GenerateSetsOfThree` outputs a sequence of `Lists`, each of which contains three positive integers. The sequence is to be terminated by a `List` comprising [-1, -1, -1].

What change is required to output objects containing six integers?
How could you parameterise this in the system?