GPars - Groovy Parallel Systems

Jon Kerridge

Version 1.0, 2015-10-29

Table of Contents

Basic Concepts 1

  Process 1

  Channel 2

  Timers 3

  Alternatives 3

Summary 4

Samples 5

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.

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.

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 writes to the channel and the other reads 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.

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.

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.

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.

Samples

ConsumeHN.groovy

```
// GPars (formerly GParallelizer)
//
// Copyright © 2008-10  The original author or authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package c2

import org.jcsp.lang.*


class ConsumeHN implements CSProcess {

    def ChannelInput inChannel

    void run() {
        def first = inChannel.read()
        def second = inChannel.read()
        println "\n${first} ${second}!\n"
    }
}
```