Groovy Parallel Patterns: A Library to Support Parallelization

Jon Kerridge, Kevin Chalmers and Jan Bækgaard Pedersen*

School of Computing, Edinburgh Napier University, UK

*School of Computer Science, University of Nevada, Las Vegas, USA

j.kerridge@napier.ac.uk, k.chalmers@napier.ac.uk and matt.pedersen@unlv.edu

Abstract. Much has been written concerning the use of Parallel Design Patterns and Skeletal Frameworks but few, if any, have then converted the ideas contained into software products that can be used in the deployment of such parallel systems. The Patterns and Skeletons so far proposed still require the programmer to have a great deal of understanding of parallel systems design. The Communicating Process Architecture community has been producing reusable plug and play components that can be used to build parallel systems based upon Hoare’s Communicating Sequential Processes (CSP) theory for more than 30 years using various languages. These ideas are now integrated into the Library described in this paper.

Keywords. Build and Deployment, Parallelization, Patterns, Skeletons

Introduction

Skeletal frameworks provide simple methods to develop parallel software. They are aimed at domain experts who are not parallel programmers. Skeletal frameworks do this by providing mechanisms for building parallel applications through reusable blocks. These ideas have become important as multi- and many-core architectures became the norm.

The Communicating Process Architectures (CPA) community (www.wotug.org) has been developing parallel applications for over 30 years. The CPA manifesto focuses on communication between processes via channels (message passing). The nature of CPA promotes the same block ideas that skeletal frameworks do. The CPA community have developed plug and play libraries as reusable blocks for common problems and communication patterns. Many of these plug and play blocks share similarities with common skeletal blocks.

In this paper we present work examining the communication semantics of algorithmic skeletons. Our work is built on the RISC-pb'l [4, 1] block specifications, extending these specifications by exploring the different communication patterns. The contribution of our work is the creation of a library of reusable components that enable build and deployment of parallel systems in which the required algorithms are contained within a small number of serial objects. The programmer simply has to deploy these algorithms within a parallel structure that can be easily invoked.

We present the rest of our paper as follows. In Section 1 we provide a brief review of the current state of the art. In Section 2 the design goals of the library are presented. In Section 3, we present a simple example of the use of the library. Section 4 presents a more detailed overview of the library, showing how lower level processes are incorporated into higher level ones. Sections 5 and 6 show how the basic components of the library can be used to build reusable skeletons and patterns. Section 7 introduces a notation that
simplifies the design of solutions. This notation is used in Section 8 to explore a more complex example based on the construction of a concordance for a large text. Section 9 discusses the performance of the architectures presented in the paper. Finally, some conclusions are drawn together with activities for further work.

1. Background and Related Work

Algorithmic skeletons are a technique for non-parallel programmers (domain experts) to exploit parallelism. An example skeleton is a pipeline which provides a template into which functions can be placed by the programmer. A number of such skeleton libraries exist; eSkel [3], Muesli [2], Skandum [6], and SkeTo [7]. Gonzalez-Velez [5] surveyed the skeletal libraries in 2010.

More recently, development of description languages for skeleton programming has been undertaken. A description language allows the programmer to detail the structure of their application as a collection of components interacting with each other. This work builds on Danelutto et. al's [4] RISC-pb specifications. RISC-pb [4, 1] approaches the problem of a design language by producing a limited set of general purpose building blocks. These blocks are divided into three types; wrappers, combinators, and functionals. Each type supports a different part of the parallel solution. These concepts are explored further in [8].

Similarly, descriptions of parallel patterns are widely available [9, 10], that show the programmer how they can go about designing a solution to the problem provided they have some understanding of how the different architectural levels are deployed. These patterns require the programmer to have some understanding of the underlying target parallel architecture. This means the programmer is already more skilled than a programmer just wanting to exploit parallelism.

The Groovy Parallel Patterns (GPP) Library builds upon an underlying library, JCSP (Communicating Sequential Processes for Java), originally developed at the University of Kent [11]. It also utilises a further library Groovy Parallel [12] designed to make utilisation of JCSP much simpler. The use of these libraries is discussed in [13, 14]. The libraries are also available as part of the GPars project [15].

2. Design Aims of GPP

The primary goal of GPP is to provide a set of foundation processes that can be easily ‘plugged’ together to form a process network that can be used to solve a specific problem. The library also contains some basic patterns and skeletons constructed from the foundation processes which simply require the programmer to build the required problem class definitions. The foundation processes, skeletons and patterns are all designed to terminate in a well-defined manner such that on completion all the resources used by the network are recovered. GPP has been designed so that the processes can be used in multi-core and cluster configurations, with no change to the definition of the processes. In the case of cluster based solutions the processes on each node of the network have to be invoked separately.

The classes that provide the algorithms for the problem solution contain no parallel content. The solution classes are required to implement a specific set of methods that are accessed from the foundation processes. These methods are defined in the Data Class Interface. A further class DataClass is defined that contains a null implementation of all the methods defined in the interface. All user defined classes extend DataClass.
3. A First Example – Monte Carlo-π

The value of $\pi$ can be approximated using Monte Carlo methods [15]. If random points are generated in the unit upper-quadrant square then the ratio of points within the unit radius circle to all the points created is $\pi/4$. The more random points the better the approximation. Listing 1 gives the script of a process that does this calculation using any number of workers (synonymously cores) and the corresponding console output in Output 1

```java
int workers = 4
int instances = 1024
int iterations = 100000

println "RunMCpiFarmPattern workers = $workers,
instances = $instances,
iterations per instance = $iterations"

def farming = new DataParallelCollect (
dataClassName: MCpiData.getName(),
resultsClassName: MCpiResults.getName(),
initialData: [instances],
createData: [iterations],
workers: workers,
workerOp: MCpiData.opCode)
farming.run()
```

Listing 1 The Monte Carlo-π Script

RunMCpiFarmPattern workers = 4,
instances = 1024,
iterations per instance = 100000
Final Result: iterations = 102400000,
within = 80423303, pi = 3.1415352821350098
Math value of pi is 3.141592653589793
Collected 1024 Results
Time taken = 6334 milliseconds

Output 1 Output from the Monte Carlo-π Script

3.1 Discussion of the Monte Carlo-π Script

The script defines three values used in the calculation. The value of workers specifies the number of cores to be used as this solution runs on a single multi-core platform; instances specifies the number of MCpiData objects that are to be written into the process network and iterations will be used to initialise those objects with the number of random points each object is to generate. The final output is generated using the MCpiResults object.

The object DataParallelCollect creates an instance of a data parallel network with as many internal parallel processes as the value of workers. Each instance of the internal processes will undertake the function indicated by opCode. Finally, the network contains an instance of a process that collects the result using the MCpiResult class. The network is executed by calling the run method of the object farming, which has the effect of calling all the run methods of all the internal processes.

3.2 MCpiData

MCpiData has the structure shown in Listing 2. The method initClass is used to initialise the static and any other variable of the class using values contained in the parameter p that
are passed as the property `initialData` in the object `farming`. The method `createInstance` is used to initialise properties of each instance of `MCpiData` that is created. The parameter `d` is passed as the property `createData` in the object `farming`. The method `operation` defines the function to be carried out by each worker process in farm network. The method `invoke` provides the means whereby worker processes can identify the function to be undertaken and then call the required operation method with the required parameters. Each method returns an appropriate constant value defined in `DataClassInterface`.

```java
class MCpiData extends org.jcsp.gpp.DataClass {
    int iterations = 0
    int within = 0
    static int instance = 1
    static int instances
    static final int errorState = -1
    static final int opCode = 1

    int initClass (List p) {
        instances = p[0]
        return completedOK
    }

    int createInstance (List d){
        if ( instance > instances) return normalTermination
        else {
            iterations = d[0]
            within = 0
            instance = instance + 1
            return normalContinuation
        }
    }

    def operation(){
        def rng = new Random()
        float x, y
        for ( i in 1 .. iterations){
            x = rng.nextFloat()
            y = rng.nextFloat()
            if ( ((x*x) + (y*y)) <= 1.0 ) within = within + 1
        }
        return completedOK
    }

    int invoke (int fn,  List l ){
        switch (fn){
            case opCode:
                return operation()
            break
            default:
                return errorState
        }
    }
}
```

Listing 2 The MCpiData Class Definition

### 3.3 MCpiResults

Listing 3 shows the definition of the `MCpiResults` class. The method `finalise` is used to undertake final output of the results produced by the network. The method `collector` is used to evaluate each the results from the incoming data objects passed to the `Collect` process, which calls these methods.
class MCpiResults extends org.jcsp.gpp.DataClass {
    static int iterationSum = 0
    static int withinSum = 0

    int finalise(List p) {
        def pi = 4.0 * ((float) withinSum / (float) iterationSum)
        println """Final Result: iterations = $iterationSum,
        within = withinSum, pi = $pi"
        println "Math value of pi is ${Math.PI}"
        return completedOK
    }

    int collector(def o) {
        iterationSum = iterationSum + o.iterations
        withinSum = withinSum + o.within
        return completedOK
    }

    int initClass (List d){
        return completedOK
    }
}

Listing 3 The Definition of the Class MCpiResults

The above discussion has demonstrated that the programmer is solely concerned with writing sequential methods that will be called as required within the processes that make up GPP. The next section discusses the structure of the library.

4. Structure of the GPP Library

The library, org.jcsp.gpp, comprises a number of packages containing processes that can be connected together to form larger networks that create parallel architectures. The description will commence with the lowest level processes and progress to the higher level ones.

4.1 Package workers

A Worker process provides the means by which a function can be invoked. There are two versions; terminating and non-terminating versions. Apart from termination the behaviour is similar and is shown in Listing 4, where input and output are the channels from the process reads and writes respectively. All other parameters are passed as properties of the class. The object WorkerClass, can be null, but if present provides some local data storage and methods used by the algorithm. Typical processing outputs each input object to the next stage in the process network but an option exists whereby the content of the WorkerClass is output once all input data objects, o, have been read.

    workerClass.initClass(workerInitData)
    while (running) {
        o = input.read()
        o.invoke(workerOp,[dataModifier, workerClass])
        output.write(o)
    }

Listing 4 Worker Process Behaviour

The process terminates when a special Termination object is read and sets running false.
4.2 Package terminals

The primary processes in the terminals packages deal with the creation of data objects by an Emit process and the collection of results by a Collect process. The behaviour of the Emit process is shown in Listing 5, where the name of the EmitClass is passed as a property of the class as are initialData and createData. Created objects are written to the channel output.

```java
emitClass.newInstance()
emitClass.initClass(initialData)
while (emitClass.createInstance(createData) == normalContinuation) {
    output.write(emitClass)
    emitClass.newInstance()
}
```

Listing 5 Emit Process Behaviour

The behaviour of the Collect process is shown in Listing 6, where the name of the ResultsClass is passed as a property to the class. Input data objects are read as o, from the input channel. The methods collector and finalise have to be defined in the ResultsClass. The remaining parameters are passed as properties of the Collect class.

```java
resultsClass.initClass(initData)
o = input.read()
while ( o != UniversalTerminator ){
    resultClass.collector(o)
    o = input.read()
}
resultsClass.finalise(finaliseData)
```

Listing 6 Collect Process Behaviour

The ResultsClass is used as local storage to collect the results and its content is only output by the finalise method once all the input data objects have been read.

It can be observed that in all the processes discussed so far whenever an object is read from the process’ input channel the type of the object is not required. This in fact is the case with all processes in the library. The input object is always calling one of its own methods (e.g. invoke) or is passed as a parameter to another object which the user has also defined and hence the type of the object is known at design time.

4.3 Package connectors

Two types of connector components are provided; spreaders and reducers. Spreaders take an input object and then transfer the object to one of more processes, typically Worker processes. If the same object is to be transferred to more than one subsequent process then the object is cloned either using the default clone method or if necessary an overridden clone method that creates a deep copy of the object.

4.3.1 Spreaders

The processes are supplied in a number of different variations depending on the nature of the channel connections provided by the process as follows:  
Any expects the any end of a channel to which many process can read or write but only one input object is processed on each iteration of the process  
List expects a channel list, either input or output
One expects a single channel providing a connection to one process
Requested expects a channel pair, one requesting data, the other to transfer the data
Requesting the opposite end to a Requested end; used in cluster based networks

The nature of the process is defined by the central part of the name:
Fan processes one object at a time and in the case of a List output will write the
object to the next list out channel end in sequence
SeqCast outputs a single input value to all the outputs in sequence
ParCast outputs a single input value to all the outputs in parallel

Table 1 shows the process names and behaviours of the spreaders package, where the
parameter destinations is the number of processes connected to an outputAny channel.
Writing to an any channel means that the output object can be read by any of the inactive
processes waiting to read from the channel. If all the reading processes are busy then the
written objects are queued in a FIFO data structure. The italicised abbreviations are used to
identify the processes in the notation described in Section 7.

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneFanAny (input, outputAny)</td>
<td>//ofa</td>
</tr>
<tr>
<td>AnyFanAny (inputAny, outputAny)</td>
<td>//afa</td>
</tr>
<tr>
<td>OneFanList (input, outputList)</td>
<td>//olf</td>
</tr>
<tr>
<td>AnySeqCastAny (inputAny, outputAny, destinations)</td>
<td>//asa</td>
</tr>
<tr>
<td>OneParCastList (input, outputList)</td>
<td>//opl</td>
</tr>
<tr>
<td>OneSeqCastList (input, outputList)</td>
<td>//osl</td>
</tr>
<tr>
<td>OneFanRequestedAny(input, inRequestAny, outResponseAny)</td>
<td>//orf</td>
</tr>
<tr>
<td>RequestingFanAny (outputAny, outRequestAny, inResponseAny)</td>
<td>//rfa</td>
</tr>
</tbody>
</table>

Table 1 Processes and Behaviours of some of the Spreaders Package Processes

The methods broadcast and broadcastSeq are provided by Groovy Parallel. They
both output a clone of the object passed as a parameter to each element of the outputList;
broadcast achieves this using a parallel write operation whereas broadcastSeq writes to
each element in sequence. The fact that objects are cloned, using a deep copy, is an
important principle of the GPP Library because it means that within a single node all
objects are unique and thus it is safe to pass object references from one process to another.
When an object is written from one process and read by another process the object
reference is passed between the processes. If we could not guarantee that all objects are
distinct then we could have more than one process accessing the same object reference at the same time and this would lead to problems with mutual exclusion between processes. The library has been constructed to ensure that such problems do not occur, however, users have to be aware that the default clone method is insufficient if the object being passed between processes contains further objects. In this case the user has to override the default clone method with one that undertakes the deep copy of the object.

4.3.2 Reducers

Table 2 shows the process names and behaviours of processes from the package reducers. These use the same naming conventions as the spreaders described previously. In all cases the processes output to a single channel but input from many processes connected by either an any channel or a list of channels. Reading from an any channel means that as soon as one of the processes writing to the channel attempts to write then the reading process will read that object unless it is already dealing with a previous input in which case the write request is queued in a FIFO structure and means that reads are processed in the order the writes occurred.

<table>
<thead>
<tr>
<th>AnyFanOne(output, inputAny) //afo</th>
<th>ListFanOne(output, inputList) //lfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (true) {</td>
<td>alt = new ALT (inputList)</td>
</tr>
<tr>
<td>output.write(inputAny.read())</td>
<td></td>
</tr>
<tr>
<td>i = alt.fairSelect()</td>
<td>while (true) {</td>
</tr>
<tr>
<td>output.write(inputList[i].read())</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>values = inputList.readSeq()</td>
</tr>
<tr>
<td></td>
<td>for ( i in 0 .. inputList.size()</td>
</tr>
<tr>
<td></td>
<td>output.write(values[i])</td>
</tr>
<tr>
<td></td>
<td>for ( i in 0 .. inputList.size()</td>
</tr>
<tr>
<td></td>
<td>output.write(values[i])</td>
</tr>
</tbody>
</table>
| ListParOne(output, inputList) //lpo | ListSeqOne(output, inputList) //ls
| while (true) {                   | while (true) {                    |
|       values = inputList.read()  |       values = inputList.readSeq() |
|       for ( i in 0 .. inputList.size() |
|         output.write(values[i])  |       for ( i in 0 .. inputList.size() |
| }                                 |         output.write(values[i])  |

Table 2 Processes and Behaviours of theReducers Package Processes

The ALT construct, from Groovy Parallel, creates a means whereby the readiness of any of the elements in the input List can be assessed. The method fairSelect returns the index of the inputList element that is chosen such that all elements are given an equal share of the available bandwidth. If no element is ready then fairSelect waits until one is ready and then processes the associated data. If one element is ready then that element is processed. If more than one is ready then the element that has been least accessed in prior iterations is chosen. An ALT also has other selection mechanisms; priSelect, which choses the first element in sequence and select, which chooses the element randomly, when more than one element is ready to communicate.

The method read applied to a channel input list reads all the channels in parallel and returns a list of values the same size as the input list. Similarly, readSeq has the same outcome but reads the elements of the input list in sequence.

We now have completed the description of the lowest level components of the library. We could leave the library at this lowest level because we can construct any process network using the components described so far, however that would make using it very difficult and error prone. These higher level components are described in the next section.
5. Skeletons

Two basic skeleton packages are provided, groups and pipelines together with a composite package defining combinations of the other two packages.

5.1 Groups

Process networks within groups provide that capability associated with a parallel-for statement by creating a parallel of Worker or WorkerTerminating processes. They are typically used in data parallel applications where the same algorithm is applied to many instances of the same data. The different types of group reflect the nature of the channel connections they have for input and output; any or channel lists. Thus the package provides options for: AnyGroupAny, AnyGroupList, ListGroupList, ListGroupAny and a special group ListGroupCollect which contains a parallel of Collect processes. The number of parallel processes is specified by a property in the process’ constructor.

As an example of the code required to create a group, see Listing 7, which shows the code used to form an AnyGroupAny process.

```groovy
class AnyGroupAny implements CSProcess{
    ChannelInput inputAny
    ChannelOutput outputAny
    String workerClassName = null
    list workerInitData = null
    int operation
    list modifier = null
    int workers
    boolean outData = true

    void run() {
        List network = (0 .. < workers).collect {e ->
            new WorkerTerminating ( input: inputAny,
                output: outputAny,
                workerClassName: workerClassName,
                workerInitData: workerInitData == null ? null : workerInitData[e],
                workerOp: operation,
                dataModifier : modifier == null ? null : modifier[e],
                outData: outData )
        }
        new PAR (network).run()
    }
}
```

Listing 7 Definition of the AnyGroupAny Process

As can be observed, properties that are not specified, default to a null value. A process network is created using a Groovy collect closure. The PAR class creates a process network object that can be run and is obtained from the Groovy Parallel Library. The property outData is used to determine whether the input data objects are output as they are processed or whether a single output of the worker class is output once all the input objects have been processed.

5.2 Pipelines

The package pipelines contains two processes; OnePipelineOne and OnePipelineCollect. These processes are typically used in task parallel situations where a data object is passed through a series of processes (stages) each of which can be executed in parallel. The only difference is that in a OnePipelineCollect the last stage of the
pipeline is a Collect process. Listing 8 shows the way in which a OnePipelineCollect process is created, omitting all the properties from the constructors.

class OnePipelineCollect implements CSProcess{
    ChannelInput input
    int stages = 2
    String [] workerClassName = null
    String resultClassName
    List workerInitData = null
    List stageOp = []
    List stageModifier = null
    List resultInitData = null
    List finaliseData = null
    boolean [] outData = null

    void run() {
        if (stages < 2) unexpectedReturnCode("OnePipelineOne: insufficient worker stages ", stages)
        if (outData == null) outData = (0..<stages).collect{ i -> return true}
        int lastIndex = stages - 1
        def interConnect = Channel.one2oneArray(stages)
        def firstStage = new WorkerTerminating( input: input,
            output: interConnect[0].out(),
            workerClassName: workerClassName == null? null : workerClassName[0],
            workerInitData: workerInitData == null ? null : workerInitData[0],
            workerOp: stageOp[0],
            dataModifier: stageModifier == null ? null : stageModifier[0],
            outData: outData[0])
        def collectStage = new Collect( input: interConnect[lastIndex].in(),
            resultClassName: resultClassName,
            initData: resultInitData,
            finaliseData: finaliseData)
        List stageProcesses = []
        for (s in 0..< lastIndex){
            stageProcesses << new WorkerTerminating( constructor as above )
        }
        stageProcesses << firstStage
        stageProcesses << collectStage
        new PAR(stageProcesses).run()
    }
}

Listing 8 Partial Definition of a OnePipelineCollect Process

Pipelines always process a single input channel, input. The definition uses a WorkerTerminating process so that it terminates each stage as the last input data object is processed. A pipeline must always have at least two stages one of which is the Collect process in the case of Listing 8. The firstStage is defined specifically as is the collectStage. The remaining stages are created as stageProcesses to which the other stages are appended.

5.3 Composites

The composites package defines process networks that are either a pipeline of groups or a group of pipelines. These processes networks are characterized by the number of workers in each group and the number of pipeline stages. Any group stage that has a List type output can also create a synchronising barrier that ensures that all processes in the parallel complete the process before any of them can write their output to the next process as might happen in composite processes.
6. Patterns

The `patterns` package defines commonly used parallel architectures, such as Data Parallel and Task Parallel. Thus the pattern `DataParallelCollect` used in the Monte Carlo-π script (Listing 1, Section 3) is defined as shown in Listing 9.

```groovy
class DataParallelCollect {
    String dataClassName, resultsClassName
    String workerClassName = null
    List workerInitData = null
    List initialData = null
    List createData = null
    int workers, workerOp
    boolean outData = true

def run = {
    def toFanOut = Channel.one2one()
    def toFarm = Channel.one2any()
    def fromFarm = Channel.any2one()
    def collectChan = Channel.one2one()

    def emitter = new Emit( output: toFanOut.out(),
        emitClassName: dataClassName,
        initialData: initialData,
        createData: createData)

    def fanOut = new OneFanAny(input: toFanOut.in(),
        outputAny: toFarm.out(),
        destinations: workers)

    def farmer = new AnyGroupAny( inputAny: toFarm.in(),
        outputAny: fromFarm.out(),
        workerClassName: workerClassName,
        workerInitData: workerInitData,
        workers: workers,
        operation: workerOp,
        outData: outData )

    def fanIn = new AnyFanOne(inputAny: fromFarm.in(),
        output: collectChan.out(),
        sources: workers)

    def collector = new Collect( input: collectChan.in(),
        resultClassName: resultsClassName )

    new PAR([emitter, fanOut, farmer, fanIn, collector]).run()
}
```

Listing 9 DataParallelCollect Definition

Initially, the channels needed to connect the processes together are defined, followed by instances of the processes `Emit`, `OneFanAny`, `AnyGroupAny`, `AnyFanOne`, and `Collect`. These are then formed into a parallel network and run. Thus the simple use of the pattern in Listing 1 results in the parallel composition of five supporting processes plus as many worker processes as the value of the property `workers`. The programmer is only concerned with the algorithms needed to create the solution and does not have to be aware of the intricacies associated with the creation of the parallel architecture. Obviously, if the architecture has to be specially created then the programmer may have to use members of the composites and in some cases groups and pipelines to create the required parallel network. This is a more complex task but follows easily from a specification of the required processing stages. This can be aided by the use of the notation described in the
next section.

7. Notation to Support GPP

The network shown in Listing 9 can be represented by the representation shown in Structure 1.

\[ \text{emit} \rightarrow \text{of} \rightarrow [f]_n \rightarrow \text{afo} \rightarrow \text{collect} \]

**Structure 1 Structure of the DataParallelCollect Network**

The structure describes a sequence of processes connected from left to right where:

- \( \text{emit} \) is the name or an abbreviation of a process in the library
- \( \rightarrow \) means the process is connected to the next process by one or more channels
- \( [f]_n \) means \( n \) parallel processes undertaking the function \( f \)

The nature of the channel connections required can be deduced from the nature of the process that connects the parallel components together where \( a \) means any, \( o \) means one and \( l \) means list. The nature of the connector is described by the letters \( f \) (Fan), \( p \) (ParCast) and \( s \) (SeqCast). Structure 2 shows a different formulation of the above architecture where the parallel group is replaced by a group that has a worker class, which collects intermediate results which are only output once all the data objects have been emitted.

\[ \text{emit} \rightarrow \text{of} \rightarrow [f]_n \rightarrow \text{afo} \rightarrow \text{collect} \]

**Structure 2 Alternative Invocation of the DataParallelCollect Network**

8. Concordance

Creating a concordance of large piece of text is often used as an example where parallel processing can improve performance. The goal is to produce an output which shows where the same string of words occurs throughout the text file. This should be achieved for strings of words of length 1 to some value \( N \). Processing can be broken down into five stages.

**Stage 1** Processing word strings and in particular comparing them is complex and time consuming so we shall extract all the words from the text, removing unnecessary punctuation, and then calculate a corresponding integer value for each word based upon summing the letter codes for the word. We shall store these words and the list of word values.

**Stage 2** creates the sums of sequence of values for strings of length 1 to \( N \), which generate \( N \) valueLists.

**Stage 3** creates \( N \) maps each of which comprises a key based on a value from a valueList and an entry that contains all the locations where that value occurs. These maps are called indicesMap.

**Stage 4** disambiguates indicesMap because a key value may refer to different word sequences. Thus for each key in an indicesMap we extract the word sequence to which it corresponds, recall that the entry contains the location of the value in the text and we have stored the words. Thus we can build up a map comprising a key of a word string together with an entry comprising the locations of that specific
word string. We shall call the N maps the wordsMap.

Stage 5 outputs the wordsMap in a human readable form ensuring that only strings that are repeated at least the minimum number of times are output.

Stage 1 can be implemented in the Emit process and stage 5 can be implemented the Collect process. We thus can build a group of pipelines or a group of pipelines to implement stages 2, 3 and 4. The two architectures are shown in Structure 3 where $s_i$ represents the stage to be evaluated.

$$emit \rightarrow ofa \rightarrow |s_2|_n \rightarrow |s_3|_n \rightarrow |s_4|_n \rightarrow afo \rightarrow collect$$

Structure 3 Simple variants of the Concordance Architecture

Structure S3a represents a pipeline of groups with n parallel workers, whereas S3b is a group of n pipelines.

We can also modify the initial input phase in the emit stage because this involves many disc accesses, especially for large input files. We modify emit so that it reads a number of words into a block and then sends that to a worker that does the removal of punctuation and the initial calculation of the integer value corresponding to each word. We then have to provide a means of recombining the blocks into one large block before then emitting the required instances into the remainder of the network. We deploy a group and have a different fan out and fan in mechanism that sends work to each worker in the group in sequence, which can be achieved using OneFanList and ListFanOne connectors. Thus the requirement is to balance the load so that the time it takes to read in a block from file is the same as it takes all the workers to process a block. The revised emit stage is shown in Structure 4, where $w$ parallel instances of the function $s_1$ to remove punctuation and undertakes the initial summing. The process combine takes the blocks in turn and recombines them into a single block that can then be emitted to either of the previous networks. The process emitInput reads in some input data and then emits instances as required. The additional processes referred to in the description already exist within the GPP Library.

$$emit \rightarrow ofl \rightarrow |s_1|_w \rightarrow lfo \rightarrow combine \rightarrow emitInput \rightarrow$$

Structure 4 Alternative Definition of the emit Stage in Structure 3

9. Performance Evaluation

One advantage of the GPP design philosophy is that the programmer can develop the required algorithmic components with no regard for the fact that these are going to be executed in a parallel environment. In fact, it means that the algorithms can be tested in a purely sequential environment prior to parallelisation, simply by calling the required methods in sequence. This means that it is relatively easy to determine the speedup achieved by the parallel solution. In addition we can also run the parallel solution with just one parallel worker, which will give us some indication of the cost of installing the parallel infrastructure because in most cases the setup cost of the parallel architecture will be measurable. We can then undertake the processing using different numbers of workers up to some limit imposed by the number of available cores. All the experiments were run on
an Intel Core2 Quad Q8400 processor running at 2.67GHz with 8 GB memory using 64 bit software systems. The applications were invoked from Eclipse in a Windows 10 environment. The performance is shown as the median value of 10 runs of each architectural variant.

9.1 Monte Carlo π

Chart 1 shows four variants of the architecture; the two parallel architectures corresponding to Structures 1 and 2 shown in Section 7, together with two sequential versions. The Sequential version creates 1024 objects each of which does 10000 iterations, whereas SeqUnrolled creates a single object containing 102400000 iterations. We would expect that latter of these to be faster and it is by 400 milliseconds. Both of the parallel solutions are slower with one worker that either of the sequential solutions but that is not surprising given the parallel infrastructure has to be created. Perhaps the difference between them of more than 8 seconds is surprising!

Chart 1 Relative Performance of the Monte Carlo π Architecture Variants

The two parallel solutions have a similar general pattern in terms of the reduction in time with the version that has an additional worker class being consistently worse. There is a reasonable benefit of moving from 1 to 2 workers, a speedup of 1.54, but after that speedup is not that linear. This can be explained by the fact that even though we have the specified number of worker, there are an additional 4 processes (emit, ofa, afo, collect) running on the processor. Thus the four cores are already being shared by more parallels than cores and hence threads will be swapped out, hence the reduction we see is perhaps not that surprising.

9.2 Concordance

Chart 2 shows variants of the architecture for those options shown in Structure 3 (Section 8). The version shown as Structure 4 was implemented but found to have a much worse performance and thus ignored from discussion. The text used was the bible
searching for strings of length \( N = 6 \) where the repeated string occurs at least twice.

Unlike the Monte Carlo \( \pi \) example, Concordance, immediately gives an improvement for the parallel solution where the value of \( n \) in S3a and S3b is shown on the x-axis. As can be seen, the increase in \( n \) does yield a small improvement in performance but, yet again, not as much as might be expected because there are a number of other processes running in parallel. With \( n = 1 \) there are 7 running processes and \( n = 4 \) there are 16 processes. Undoubtedly, the processing time is mostly influenced by the amount of input-output that is required. The input text is 4.6 MB and the total output is 26.1 MB in six files. Even though the file handlers run in parallel there is a likelihood that disc access may be having a major effect. A more detailed analysis of the timing associated with each stage of the algorithm shows that the total time spent in each of the stage \( S_2 \) to \( S_4 \) is relatively small.

10. Conclusions and Further Work

A library has been developed that allows a programmer to create a parallel solution to a problem without having to construct the parallel architecture itself. The programmer only needs to be concerned with the detail of the solution algorithm. However, the algorithm does have to take account of parallel requirements; in particular, no data structure should be written to by more than one process at a time. The library ensures that as objects are transferred from one process to another there is no possibility that the processes can refer to the same object reference. At the same time copies of objects are not made unless strictly necessary, thereby saving memory.

This paper has not considered the requirements of cluster processing. In [14] a means of loading process networks over a TCP/IP connected cluster is described. Future work will reflect on that architecture to make it as simple as possible to distribute a solution over a cluster.
Code Availability

The code of the Groovy Parallel Patterns Library is available at https://bitbucket.org/jkerridge/org_jcsp_gpp/downloads where the required (JCSP and Groovy Parallel) libraries are also held.

References


Outline of the Presentation

The presentation will cover the material presented in this paper without going into the code in as much detail. It will also present solutions to further parallel problems including prime numbers and the Goldbach conjecture and an embarrassingly parallel problem such as Mandelbrot.