Experiments in Multicore and Distributed Parallel Processing using JCSP

Jon KERRIDGE
School of Computing, Edinburgh Napier University, Edinburgh UK, EH10 5DT
j.kerridge@napier.ac.uk

Abstract. It is currently very difficult to purchase any form of computer system be it, notebook, laptop, desktop server or high performance computing system that does not contain a multicore processor. Yet the designers of applications, in general, have very little experience and knowledge of how to exploit this capability. Recently, the Scottish Informatics and Computer Science Alliance (SICSA) issued a challenge to investigate the ability of developers to parallelise a simple Concordance algorithm. Ongoing work had also shown that the use of multicore processors for applications that have internal parallelism is not as straightforward as might be imagined. Two applications are considered: calculating π using Monte Carlo methods and the SICSA Concordance application. The ease with which parallelism can be extracted from a single application using both single multicore processors and distributed networks of such multicore processors is investigated. It is shown that naïve application of parallel programming techniques does not produce the desired results and that considerable care has to be taken if multicore systems are to result in improved performance. Meanwhile the use of distributed systems tends to produce more predictable and reasonable benefits resulting from parallelisation of applications.

Keywords: multicore processors, distributed processing, parallel programming, Groovy, JCSP, Monte Carlo methods, concordance.

Introduction

The common availability of systems that use multicore processors is such that it is now nearly impossible to buy any form of end-user computer system that does not contain a multicore processor. However, the effective use of such multicore systems to solve a single large problem is sufficiently challenging that SICSA, the Scottish Informatics and Computer Science Alliance, recently posed a challenge to evaluate different approaches to parallelisation for a concordance problem. There will be other challenges to follow. The concordance problem is essentially input/output bound and thus poses particular problems for parallelisation. As a means of comparison, a simple compute bound problem is also used as an experimental framework: namely the calculation of π using a Monte Carlo method.

The aim of the experiments reported in this paper is to investigate simple parallelisation approaches (using the JCSP packages [1, 2] for Java, running on a variety of Windows platforms) and see whether they provide any improvement in performance over a sequential solution. In other words, is parallelisation worth the effort? In section 2, experiments using the Monte Carlo calculation of π are presented. Section 3 describes and discusses the experiments undertaken with the concordance example. Finally, some conclusions are drawn.

1. Calculating π Using Monte Carlo Methods

The calculation of π using Monte Carlo statistical methods provides an approximation based on the relation of the area of a square to an inscribed circle [3]. Given a circle of radius *r* inscribed in a square of side 2*r,* the areas of the circle and square are, respectively, $\pi r^2$and $4r^2$ – so, the ratio of these areas is π/4. Hence, if sufficient random points are selected within the square, approximately π/4 of the points should lie within the circle.

The algorithm proceeds by selecting a large number of points (*N = 1,024,000*) at random and determining how many lie within the inscribed circle (*M*). Thus if sufficient points are chosen, π can be approximated by *(M/N)**4. The following sequential algorithm, Listing 1, written in Groovy [4], captures the method assuming a value of *r* = 1 (and using only the top-right quadrant of the circle). The algorithm is repeated 10 times and the results, including timings, are averaged.

```
01  def r = new Random()
02  def timer = new CSTimer()
03  def pi = 0
04  def int N = 10240000
05  def startTime = timer.read()
06  for ( run in 1..10) {
07    print "-"
08    def int M = 0
09    for ( i in 1..N){
10      def x = r.nextDouble()
11      def y = r.nextDouble()
12      if (( (x*x) + (y*y)) < 1.0 ) M = M + 1
13    }
14    pi = pi +  ((double)M)/ ((double)N) * 4.0
15  }
16  def endTime = timer.read()
17  def elapsedTime = (endTime - startTime)/10
18  pi = pi / 10.0
19  println "\n$pi,$elapsedTime"
```

Listing 1. Sequential implementation of π estimation.

The 'obvious' way to parallelise this algorithm is to split the task over a number of workers (*W*), such that each worker undertakes *N*/*W* iterations. A manager process is needed to initiate each worker and collate the results when all the workers have completed their task. Listing 2 shows the definition of such a worker process using Groovy Parallel and JCSP.

```
20  class Worker implements CSProcess {
21    def ChannelInput inChannel
22    def ChannelOutput outChannel
23
24    void run(){
25      def r = new Random()
26      for ( run in 1..10){
27        def N = inChannel.read()
28        def int M = 0
29        for ( i in 1..N){
30          def x = r.nextDouble()
31          def y = r.nextDouble()
32          if (( (x*x) + (y*y)) < 1.0 ) M = M + 1
33        }
34        outChannel.write (M)
35      }
36    }
37  }
```

Listing 2. Worker process definition.

The corresponding manager process is shown in Listing 3. Each run of the calculation is initiated by a communication from the manager process to each worker {52}[1]. The manager process then waits for the returned value of *M* from each worker {53}.

```
38  class Manager implements CSProcess {
39    def ChannelOutputList outChannels
40    def ChannelInputList  inChannels
41
42    void run () {
43      def timer = new CSTimer()
44      def startTime = timer.read()
45      def workers = outChannels.size()
46      def pi = 0.0
47      def N = 10240000
48      def iterations = N / workers
49      for ( run in 1..10) {
50        print "."
51        def M = 0
52        for ( w in 0 ..< workers) outChannels[w].write (iterations)
53        for ( w in 0 ..< workers) M = M + inChannels[w].read()
54        pi = pi + ( ( ((double)M)* 4.0) / ((double)N) )
55      }
56      def endTime = timer.read()
57      def elapsedTime = (endTime - startTime)/10
58      pi = pi / 10.0
59      println "\n$workers,$pi,$elapsedTime"
60    }
61  }
```

Listing 3. Manager process definition.

This parallel formulation has the advantage that it can be executed as a single parallel within one Java Virtual Machine (JVM) or over several JVMs using net channels. Furthermore, the JVMs can be executed on one or more cores in a single machine or over several machines, simply by changing the manner of invocation.

1.1 Experimental Framework

The experiments were undertaken on a number of different machines and also over a distributed system in which each node comprised a multicore processor. Table 1 shows the three different machine types that were used.

Table 1. Specification of the experimental machines used in the experiments.

Name	CPU	cores	Speed (Ghz)	L2 Cache (MB)	RAM (GB)	Operating System	Size bits
Office	E8400	2	3.0	6	2	Windows XP	32
Home	Q8400	4	2.66	4	8	Windows 7	64
Lab	E8400	2	3.0	6	2	Windows 7	32

The Lab and Office machines were essentially the same except that the Lab machines were running under Windows 7 as opposed to XP. The Home machine was a quad core 64-bit machine. The Lab machines were also part of a distributed system connected by a 100 Mbit/sec Ethernet connected to the internet and thus liable to fluctuation depending on network traffic.

[1] The notation {n} and {n..m} refer to line numbers in one of the Listings. Each line is uniquely numbered.

1.2 Single Machine Performance

The experiments on a single machine were undertaken as follows. The sequential algorithm was executed on each machine type to determine the 'sequential' performance of each machine. The average performance for the sequential version over 10 runs for each machine type is shown in Table 2. The effect of the 64-bit architecture on the Home machine is immediately apparent. Using the Windows Task Manager to observe CPU usage on each of the machines it was noted that the maximum CPU usage was never more than 50%.

Table 2. Sequential performance of each machine.

	Office	Home	Lab
Time (secs)	4.378	2.448	4.508

The parallel version of the algorithm was then executed on each machine in a single JVM with various numbers of worker processes. The corresponding times and associated speedup is shown in Table 3. The performance in each case was monitored using the Task Manager and in each case the CPU usage was reported as 100%. However, the only version which showed any speedup of the parallel version over the sequential version was the Home machine with 2 workers. In all other cases the use of many parallel workers induced a slowdown even though the CPU was indicating a higher percentage use. The same behaviour was observed by Dickie [5] when undertaking the same Monte Carlo based calculation of π in a .NET environment. It was observed that as the number of threads increased CPU usage rose to 100% and overall completion time got worse. Further analysis using Microsoft's *Concurrency Visualizer* tool [6] showed this additional processor usage was taken up with threads being swapped.

Table 3. Parallel performance with varying number of workers in a single JVM.

Workers	Office (secs)	Speedup	Home (secs)	Speedup	Lab (secs)	Speedup
2	4.621	0.947	2.429	1.008	4.724	0.954
4	4.677	0.936	8.171	0.300	4.685	0.962
8	4.591	0.954	7.827	0.313	4.902	0.920
16	4.735	0.925	7.702	0.318	4.897	0.921
32	4.841	0.904	7.601	0.322	5.022	0.898
64	4.936	0.887	7.635	0.321	5.161	0.873
128	5.063	0.865	7.541	0.325	5.319	0.848

The Office and Lab machines use the same processor (E8400) and both show a gradual slowdown as the number of workers is increased. Whereas, the Home machine (Q8400) initially shows a speedup then followed by an initial dramatic decrease in performance which then slowly gets worse. An explanation of this could be that the L2 cache on the Q8400 is 4MB whereas the E8400 has 6MB and that this has crucially affected the overall performance.

The parallel version of the algorithm was then reconfigured to run in a number of JVMs assuming each JVM was connected by a TCP/IP based network utilising the net channel capability of JCSP. The intention in this part of the experiment was to run each

JVM on a separate core. Each JVM was initiated from the command line by a separate execution of the java environment. The experiments were conducted twice: once just using the command line **`java`** command directly and secondly using the Windows **`start`** command so that the *affinity* of the JVM to a particular core could be defined. This would, it was hoped, ensure that each JVM was associated with a distinct core thereby increasing the parallelism. In the case of the Home and Lab machines this appeared to have no effect. In the case of the Office machine an effect was observed and the execution using the **`start`** command had a similar performance to the Lab Machine. Table 4 shows the performance from runs that did not use the **`start`** command.

Table 4. Parallel performance with varying number of JVMs in a single machine.

JVMs	Office (secs)	Speedup	Home (secs)	Speedup	Lab (secs)	Speedup
2	4.517	0.969	2.195	1.115	4.369	1.032
4	4.534	0.966	1.299	1.885	4.323	1.043
8	4.501	0.973	1.362	1.797	4.326	1.042

The Office machine, which uses Windows XP showed a slowdown when run without the **`start`** command, whereas the other two machines both showed speedups, relative to the sequential solution. These machines use Windows 7 and, as there was no difference in the performance when using **`start`** or not, it can be deduced that Windows 7 *does* try to allocate new JVMs to different cores.

The Home machine has 4 cores and it can be seen that the best speedup is obtained when 4 JVMs are used. Similarly, the Lab machine has two cores and again the best speedup occurs when just two JVMs are utilised.

1.3 Distributed Performance

The multi JVM version of the algorithm was now configured to run over a number of machines using a standard 100 Mbit/sec Ethernet TCP/IP network. These experiments involved Lab machines only, which have two cores. One of the machines ran the TCPIPNode Server, the Manager process and one Worker in one core. The TCPIPNode Server is only used to set up the net channel connections at the outset of processing. The Manager is only used to initiate each Worker and then to receive the returned results and thus does not impose a heavy load on the system. The performance using both two and four machines is shown in Table 5.

Table 5. Performance using multiple JVMs on two and four machines.

	Two Machines		Four Machines	
JVMs	Time (secs)	Speedup	Time (secs)	Speedup
2	4.371	1.031		
4	2.206	2.044	2.162	2.085
8			1.229	3.668
16			1.415	3.186

The best performance is obtained when the number of JVMs used is the same as the number of available cores. Unfortunately, the best speedup relates to the number of machines and not the number of available cores.

1.4 Conclusions Resulting from the Monte Carlo π Experiments

The Monte Carlo determination of π is essentially an application that is processor bound with very little opportunity for communication. Hence the normal behaviour of CSP-based parallelism, with many processes ready to execute but awaiting communication, does not happen. JCSP currently relies on the underlying JVM to allocate and schedule its threads (that implement JCSP processes) over multiple cores. In turn, the JVM relies on the underlying operating system (Windows, in our experiments). The disappointing observation is that this combination seems to have little ability to make *effective* use of multiple cores for this kind of application. Utilising parallel processes within a single JVM had little effect and the result was worse performance. Performance improvement was only achieved when multiple machines were used in a distributed system.

2. Concordance Related Experiments

The SICSA Concordance challenge [7] was specified as follows:

> ***Given:*** *a text file containing English text in ASCII encoding and an integer N.*
>
> ***Find:*** *for all sequences, up to length N, of words occurring in the input file, the number of occurrences of this sequence in the text, together with a list of start indices. Optionally, sequences with only 1 occurrence should be omitted.*

A set of appropriate text files of various sizes was also made available, with which participants could test their solutions. A workshop was held on 13th December 2010 where a number of solutions were presented. The common feature of many of the presented solutions was that as the amount of parallelism was increased the solutions got slower. Most of the solutions adopted some form of Map-Reduce style of architecture using some form of tree data structure.

The approach presented here is somewhat different in that it uses a distributed solution and a different data structure. The use of a distributed solution using many machines was obvious from the work undertaken on Monte Carlo π. The data structures were chosen so they could be accessed in parallel, thereby enabling a single processor to progress the application using as many parallel processes as possible. However, the number of such parallel processes was kept small as it had been previously observed that increased numbers of parallel processes tended to reduce performance.

The Concordance problem is essentially input-output bound and thus a solution needs to be adopted that mitigates such effects. For example, one of the text files is that of the Bible which is 4.681 MB in size and comprises 802,300 words. For N=6 (the string length) and ignoring strings that only occur once, this produces an output file size of 26.107 MB.

2.1 Solution Approach

It was decided to use N as the basis for parallelisation of the main algorithm. The value of N was likely to be small and thus would not require a large number of parallel processes on each machine. It was thus necessary to create data structures that could read the data

structures in parallel (with each value of N accessed by a separate process). One approach to processing character strings is to convert each word to an integer value based on the sum of the ASCII values of each character in the word. This has the benefit that subsequent processing uses integer comparisons, which are much quicker than string comparisons.

The approach used to parallelise the reading of the input file was to split it into equal sized blocks, in terms of the number of words and then send each block to a worker process. The input blocks were distributed in turn over the available worker processes. Once a worker process received a block it would do some initial processing, which should be completed before the next block was to be received. This initial processing removed any punctuation from the words and then calculated the integer value of each word in the block. Some initial experiments determined that a block size of 6k words was a good compromise between the overall time taken to read the file and the ability of a worker process to complete the initial processing before the next block needed to be received so that the read process was not delayed. This appeared to be a good compromise for the number of workers being used, which were 4, 8 and 12.

The worker process could now calculate the values for N = 2..6 (N=6 was the maximum value chosen[2]). This was simply undertaken by summing the requisite number of integers in turn from the single word sequence values previously calculated during the initial phase. This could be easily parallelised because each process would need to read the N=1 values but would write to a separate data structure for N = 2..6. This was then undertaken for each block in the worker. The blocks were structured so that last N-1 words were repeated at the start of the next block. This meant that there was no need to transfer any values between workers during processing.

The second phase of the algorithm was to search each of the N sequences to find equal values, which were placed in a map comprising the value and the indices where the value was found. Only sequences with equal values could possibly be made from the same string of words. However, some values could be created from different sequences of words (simply because the sum of the characters making up the complete string was the same) and these need eliminating (see below).

This phase was repeated for each block in the worker. The result was that for each block a map structure was created which recorded the start index where sequences of equal value were found in that block. Experiments were undertaken to apply some form of hash algorithm to the creation of the value of a sequence. It was discovered that the effect was negligible in that the number of clashes remained more or less constant; the only aspect that changed was where the clashes occurred. Yet again this processing could be parallelised because each set of sequence values could be read in parallel and the resulting map could also be written in parallel as they were separated in N.

Each of these maps was then processed to determine which sequence values corresponded to different word sequences. This resulted in another map which comprised each distinct word sequence as the key and the indices where that string was found in the block. Yet again, this processing was parallelisable in N. At the end of this phase, each block contained a partial concordance for the strings it contained in a map with the sequence value as key and a further map of the word strings and indices as the entry in N distinct data structures.

The penultimate phase merged each of the partial concordances contained in each block to a concordance for the worker process as a whole. This was also parallelisable in N. The final phase was to merge to the worker concordances into a final complete concordance for each of the values of N. Initially, the sequence values in each data structure were sorted

[2] N=6 was chosen because it was known that the string *"God saw that it was good"* occurs several times in Genesis.

so that a merge operation could be undertaken with the workers sending entries in a known order to the process undertaking the merge. In the first instance the entries were sent to the initial process that read the input file where the complete concordance was created in a single file by merging the concordance entries from each worker in a manner similar to a tape merge. In a second implementation, additional processes were run in each worker that just sent the entries for one value of N to a separate merge process. There was thus N such merge processes each generating a single output file for the corresponding value of N. The effect of each of these parallelisations is considered in the following subsections.

2.2 The Effect of Phase Parallelisation

Each parallelisation did improve the performance of the application as a whole. For example, the second phase where each sequence for N = 1..6 is searched to find the indices of equal sequence values. The sequential version of the processing is shown in Listing 4.

```
62  def localEqualWordMapListN = []       // contains an element for each N value
63  for ( i in 1..N) localEqualWordMapListN[i] = []   // initialise to empty list
64   def maxLength = BL - N
65   for ( WordBlock wb in wordBlocks) {
66     // sequential version that iterates through the sequenceBlockList
67     for ( SequenceBlock sb in wb.sequenceBlockList){
68       // one sb for each value of N
69       def length = maxLength
70       def sequenceLength = sb.sequenceList.size()
71       if (sequenceLength < maxLength) length = sequenceLength   // last block
72       def equalMap = defs.extractEqualValues ( length,
73                                                wb.startIndex,
74                                                sb.sequenceList)
75       def equalWordMap = defs.extractUniqueSequences ( equalMap,
76                                                        sb.Nvalue,
77                                                        wb.startIndex,
78                                                        wb.bareWords)
79       localEqualWordMapListN[sb.Nvalue] << equalWordMap
80    }
81  }
```

Listing 4. Sequential version of equal map processing.

The data structure `localEqualWordMapListN` {62}is used to hold the map comprising the sequence value as key which has an entry, which is itself a map comprising the word string as key and the indices where the word string starts as the entry. Each of the map entries is initialised to an empty list {63}. The variables `maxLength` {64} and `length` {69..71} are used to determine how many values in the sequence values are to be used an varies with the block size and the value of N. The last block may only be partially full.

The `WordBlock` structure {65} holds all the data structures associated with each block and these are held in a list called `wordBlocks`. The loop {65..81} iterates over each such `WordBlock`. Within each WordBlock there are N `SequenceBlock`s and the loop {76..80} iterates over each of these. Each iteration initially finds the location of each sequence value that has multiple instances in the block. This is achieved by the method `extractUniqueSequences` {72} which stores the result in the map `equalMap`. The map equalMap is then passed to the method `extractUniqueSequences` {75} which creates the required output map, which is then appended to `localEqualWordMapListN` {79}. The crucial aspect of this process is that there are N `SequenceBlock`s. Thus, the process can be parallelised in N (as shown in Listings 5 and 6).

It can be seen that lines {82..85} are the same as {62..65}. Lines {86..93} create a list of process instances using the `collect` method of Groovy. The process `ExtractEqualMaps` {87} utilises the same parameters as the methods used in the sequential version. The

process list `procNet` {86} is then executed in a PAR {94}. This has the effect of determining the `localEqualWordMapListN` {82} for each value of N in parallel.

```
82  def localEqualWordMapListN = []    // contains an element for each N value
83  for ( i in 1..N) localEqualWordMapListN[i] = []
84  def maxLength = BL - N
85  for ( WordBlock wb in wordBlocks) {
86    def procNet = (1..N).collect { n ->
87      new ExtractEqualMaps( n: n, maxLength: maxLength,
88             startIndex: wb.startIndex, words: wb.bareWords,
89             sequenceList: wb.sequenceBlockList[n-1].sequenceList,
90             localMap: localEqualWordMapListN[n]) }
91    new PAR(procNet).run()
92  }
```

Listing 5. Parallel invocation of `ExtractEqualMaps`.

Listing 6 shows the definition of the process `ExtractEqualMaps`. By inspection it can be seen that the internal method calls of `extractEqualValues` {104} and `extractUniqueSequences` {106} are essentially the same as those in the sequential version except that they refer to the properties of the process rather than the actual variables. The definition is, however, unusual because it contains no channel properties. In this case the process will access memory locations that are shared between the parallel instances of the process. However the data structures were designed so that multiple processes can read the structures but they write to separate data structures ensuring there are no memory synchronisation and contention probems.

```
93  class ExtractEqualMaps implements CSProcess {
94    def n
95    def maxLength
96    def startIndex
97    def sequenceList
98    def words
99    def localMap
100   void run(){
101     def length = maxLength
102     def sequenceLength = sequenceList.size()
103     if ( sequenceLength < maxLength) length = sequenceLength
104     def equalMap = defs.extractEqualValues ( length, startIndex,
105                                              sequenceList)
106     def equalWordMap = defs.extractUniqueSequences ( equalMap,
107                                                      n, startIndex, words)
108     localMap << equalWordMap
109   }
110 }
```

Listing 6. Definition of the process `ExtractEqualMaps`.

2.3 Performance Improvements Resulting from Internal Parallelisation

Table 6 shows the performance data for the `ExtractEqualMap` phase of the algorithm. Worker Style 1 is the sequential version of the algorithm and the parallel version is represented as Style 2. The speedup resulting from increasing the number of workers is linear and is very close to the reasonable limit represented by the increased number of workers. The speedup due to the change in algorithm is about 2.5 times, which, given that N =3, is in fact for more encouraging than was achieved in the Monte Carlo π experiments. In these experiments no attempt was made to run multiple JVMs on each machine.

Table 6. Analysis of parallelisation performance by workers and technique (N=3).

Worker Style	Workers	Time (secs)	Speedup by workers	Speedup by style
1	4	138.263		
1	8	69.584	1.99	
2	4	53.600		2.58
2	8	27.559	1.94	2.52
2	12	17.957	2.98	

2.4 Effect of Parallelising the Merge Phase

Table 7 shows the total time for worker processing for a system employing 12 workers for increasing values of N. The time taken is determined by the last worker to finish its task. The workers are of Style 2 which has all internal phases parallelised. Only the merge phase is sequential. It can be seen that the ratio of the time taken is increasing more rapidly than the ratio of the file output size, thus real benefit can be achieved by overlapping the merge phase.

Table 7. Performance of the sequential merge for 12 workers.

N	Total Time (secs)	Time Ratio	Output File Size (KB)	Size Ratio
3	44.034		17,798	
4	62.044	1.41	21,412	1.20
5	82.003	1.86	23,926	1.34
6	102.896	2.34	25,810	1.45

The effect of overlapping the merge phase rather than writing all the output to a single file by undertaking N merges each of which writes to its own file is shown in Table 8. Worker Style 2 has all phases of the internal algorithm fully parallelised but writes the final concordance to file using a sequence of merges for each value of N. Whereas Worker Style 3 undertakes the merge of each value of N in parallel by having a separate merge process running on its own machine. The worker process has an internal set of N parallel processes that write the entries of each partial concordance to each of the N merge processes.

By comparing the values in Tables 7 and 8 it can be seen that the increase is now more in line with the increase in the size of the output file. The improvement is unlikely to be linear as the size of each output file varies. The largest file is associated with N = 1 because that file contains all instances of places where any single word has been repeated. In the case of the bible N=1 constitutes about 25% of the total output. The overall improvement for N = 6 in using the parallel merge represents a speed up of 1.61 and is thus worthwhile.

Table 8. Performance of the parallel merge phase for 12 workers.

Worker Style	N	Total Time (secs)	Time Ratio
2	3	44.034	
2	6	102.896	
3	3	32.319	1.36
3	6	63.866	1.61

2.5 Overall Processing Improvements

As a final experiment the performance of the system with two different input files was undertaken for N = 6 and 12 workers. This is shown in Table 9. The input file WaD contains the text for Elizabeth Gaskell's Wives and Daughters. The comparison is undertaken on the basis of the number of words in the input file, the size of the output file, the size of the file for N = 1 and the total processing time. As can be seen the smallest ratio is that for the Time, implying that as the size of the input file varies the overall time will increase at the smallest rate.

Table 9. Ratio analysis for Bible and Wives and Daughters.

	Words	Total Output (KB)	Output for N = 1 (KB)	Time (secs)
Bible	802,300	26,107	6,297	63.809
WaD	268,500	5,488	2,044	27.302
Ratio	2.99	4.76	3.08	2.34

3. Conclusions

These experiments have produced interesting and sometimes unexpected results. Perhaps the most disappointing result was that obtained in the calculation of π where it seems that, for compute bound processing, the ability of the associated JVM (supporting JCSP) and/or the operating system *automatically* to make effective use of multiple cores was limited.

The concordance example, however, produces more promising results in two ways. First the effect of introducing parallelism to the individual phases that make up the algorithm always improved the performance of the phase. Secondly, these improvements were achieved without taking any account of the fact that the machines were dual core, which is what the parallel system designer would wish. The fact that the improvements were achieved using a distributed system is also encouraging given the number of high performance clusters being built.

The key aspect of the success of the concordance solution was that it was designed parallel from the outset in terms of its internal data structures and the manner in which the processes were to communicate. A highly optimised sequential solution was not the starting point as this would have been much harder to parallelise.

The solution has the benefit of being scalable in terms of the value of N and the number of workers. It also has the benefit of being capable of scaling to any size of input file. If the available memory size in all the workers were insufficient to hold all the data structures then these could be written to file as required. Solutions that assume sufficient memory to hold the entire input file and the internal data structures are not scalable in the same manner.

Finally, the time taken to process the Bible input file, for N = 6, sequentially was 210 seconds in comparison to the 64 seconds taken by the parallel version.

Acknowledgements

The author gratefully acknowledges the very helpful comments made by the anonymous referees and also the efforts of the editors in improving the coherence and presentation of this paper.

References

[1] JCSP Home Page, `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`, accessed 28th April, 2011.

[2] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. "Integrating and Extending JCSP". In A.A. McEwan, S. Schneider, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of Concurrent Systems Engineering Series, pp. 349–370, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[3] E. Andersson. "Calculation of Pi Uisng Monte Carlo Methods", `http://www.eveandersson.com/pi/monte-carlo-circle`, accessed 28th April, 2011

[4] J Kerridge, K Barclay and J Savage. "Groovy Parallel! A Return to the Spirit of occam? ", in JJ Broenink et al (Eds.), *Communicating Process Architectures 2005*, pp. 13-28, IOS Press, Amsterdam, 2005.

[5] S Dickie. "Can design patterns (and other software engineering techniques) be effectively used to overcome concurrency and parallelism problems that occur during the development stages of video games?", *MSc Thesis*, School of Computing, Edinburgh Napier University, 2010.

[6] Microsoft, Visual Studio 2010 Concurrency Visualizer, see `http://msdn.microsoft.com/en-us/magazine/ee336027.aspx`, accessed 28th April, 2011.

[7] SICSA Concordance Challenge, `http://www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge`, accessed 28th April, 2011.