University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Distributed and Parallel Technologies**			
Coursework Number	**2**			
Deadline	**Time:**	**4:30**	**Date:**	**3/12/2018**
% Contribution to final course mark	**15%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**12**			
Submission Instructions	**On the DPT Moodle page, use the "Coursework Stage 2 Submission Assignment" to upload your Report. It is crucial that your report 1. follows the specified structure 2. provides, in appendices, your Go, C+MPI, and three Erlang Programs**			
Please Note: This Coursework cannot be Re-Assessed				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
  a. the work will be assessed in the usual way;
  b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Comparing Parallel and Distributed Programming Models:

Coursework Stage 2

BSc and MSc Students

Purpose

In this second stage of the coursework you will develop distributed programming, and experimentation skills. The parallel architecture is a shared-memory multicore. Given a sequential Erlang program, you will

- develop two concurrent versions, and a reliable version of the program
- compare the sequential and parallel performance of an Erlang program with your Go and C with OpenMP programs.
- demonstrate some reliability properties by running the reliable Erlang version against a Chaos Monkey.

Organisation

The assessed coursework work is to be carried out individually. You must develop the concurrent Erlang programs, make systematic measurements, demonstrate a reliability property, and prepare a report.

As in Stage 1, the staring program sums the Euler totient values over sequences of integers. The sequential Erlang version of the program is available from: www.dcs.gla.ac.uk/~trinder/Courses/DistrParTech/coursework/totientrange.erl

Deliverable

The deliverable is a report (including program sources) with the structure below. Reports not following this structure, or not containing all of the specified results, will be penalised.

To enable the production of **a single, coherent report on your coursework**, Sections 6 and 7 of the report can simply be reproduced from the Stage 1 coursework. Similarly, the C+OpenMP and Go performance measurements from the Stage 1 coursework can be reused in Sections 1 and 5.

The measurements are based on three data sets:

DS1: calculating the sum of totients between 1 and 15000.

DS2: calculating the sum of totients between 1 and 30000.
DS3: calculating the sum of totients between 1 and 60000.

1. You should complete the Sequential Erlang and Distributed Erlang labs before attempting this coursework.
2. You should develop and test on a lab/home machine, and only run performance measurements on a GPG cluster node. You have access to gpgnode-01, gpgnode-02, and gpgnode-03.
3. The measurements you report should be the median (middle) of three executions
4. You may find it useful to write scripts to run the measurements
5. To ensure a fair comparison all measurements should be made on a very lightly loaded GPG cluster node. Check the load on nodes using something like the unix `top` command.
6. Graphs and tables must have appropriate captions, and the axes must have appropriate labels.

Section 1 Comparative Sequential Performance (2 marks)

Complete the table below showing the runtimes of the sequential C, Erlang, and Go programs on DS1 and DS2 on a GPG cluster node. Reflect on the differences.

Data Set	C Runtimes (s)	Erlang Runtime(s)	Go Runtimes (s)
DS1			
DS2			

Table 1: Comparing C, Erlang, and Go Sequential Runtimes

Section 2 Two Worker Totient Range Erlang Program: `totientrange2Workers` (6 marks)

Develop a version of the Erlang totientrange program that spawns 3 processes: a server and two totientWorker processes.

More specifically, you should spawn and register:

A `server` process that accepts
+ A `finished` message, and terminates
+ A `{range, Lower, Upper}` message, and then
  - spawns and registers two `totientWorker` processes
  - sends messages requesting that they compute half of the range each
  - receives the range sums from the two workers, and prints the total
  - continues as a server

A `totientWorker` process that accepts
+ A `finished` message, and terminates
+ A `{range, Lower, Upper}` message, and then
  - computes the sum of the totient range
  - sends the sum to the `server`

Output from the program should resemble:

```
2> totientrange2Workers:start_server().
true
3> server ! {range, 1, 4000}.
{range,1,4000}
Worker: Computing Range 1 2000
Worker: Computing Range 2001 4000
Server: Received Sum 1216588
Server: Received Sum 3647014
Server: Sum of totients: 4863602
Worker: Finished
Worker: Finished
Server: Time taken in Secs, MicroSecs 0 806812
4>
```

In this section of the report include:
+ the output from execution of your program on ranges (1,4000), (1,15000);
+ a brief paragraph discussing any interesting features of the program (optional);
+ the source of your program (as Appendix C).

Section 3 Multi Worker Erlang Totient Range Program: `totientrangeNWorkers` (10 marks)

Develop a version of the Erlang totientrange program that spawns `NWorkers totientWorker` processes, sends requests to them to perform some fragment of the work, collects the results and prints the total.

Output from the program should resemble:

```
12> totientrangeNWorkers:start_server().
true
13> server ! {range, 1, 4000, 4}.
Worker: Started
Worker: Started
Worker: Started
Worker: Started
{range,1,4000,4}
Worker: Computing Range 1 1000
Worker: Computing Range 1001 2000
Worker: Computing Range 2001 3000
Worker: Computing Range 3001 4000
Worker: Started
Server: Received Sum 304192
Worker: Started
Server: Received Sum 912396
Worker: Started
Server: Received Sum 1519600
Worker: Started
Server: Received Sum 2127414
Server: Sum of totients: 4863602
Server: Time taken in Secs, MicroSecs 0 469226
Worker: Finished
```

```
Worker: Finished
Worker: Finished
Worker: Finished
14>
```

Hints

1. Think carefully about how the server knows that all of the `totientWorker`s have completed.
2. Think carefully about the state of the server.
3. The following function from the lecture slides may be useful for generating appropriate worker names:

```
  workerName(Num) ->
    list_to_atom( "worker" ++ integer_to_list( Num )).
```

In this section of the report include:
+ the output from execution of your program on ranges (1,4000) with 4 workers, and (1,15000) with 6 workers;
+ a brief paragraph discussing any interesting features of the program (optional);
+ the source of your program (as Appendix D).

Section 4 Reliable Multi Worker Erlang Totient Range Program:
`totientrangeNWorkersReliable` (14 marks)

Extend your `totientrangeNWorkers` program to spawn an (unsupervised) watcher process for each `totientWorker` processes.

The `watcher`<j> process should
+ spawn, register, and link to `totientWorker`<j>
+ trap exits and restart `totientWorker`<j> if required

Your program should survive being run against the following simple Chaos Monkey that kills (only) some random processes registered with names `worker1, worker2,` … `worker<NWorkers>.`

```
workerChaos(NVictims,NWorkers) ->
  lists:map(
    fun( _ ) ->
      timer:sleep(500),                %% Sleep for .5s
                                       %% Choose a random victim
      WorkerNum = rand:uniform(NWorkers),
      io:format("workerChaos killing ~p~n",
                [workerName(WorkerNum)]),
      WorkerPid = whereis(workerName(WorkerNum)),
      if                               %% Check if victim is alive
        WorkerPid == undefined ->
          io:format("workerChaos already dead: ~p~n",
                    [workerName(WorkerNum)]);
        true ->                            %% Kill Kill Kill
          exit(whereis(workerName(WorkerNum)),chaos)
      end
    end,
    lists:seq( 1, NVictims ) ).
```

Launch your server and chaos monkey with the following test harness:

```
testRobust(NWorkers,NVictims) ->
  ServerPid = whereis(server),
  if ServerPid == undefined ->
      start_server();
    true ->
      ok
    end,
  server ! {range, 1, 15000, NWorkers},
  workerChaos(NVictims,NWorkers).
```

Output from the program should resemble:

```
9> totientrangeNWorkersReliable:testRobust(4,3).
Watcher: Watching Worker worker1
Watcher: Watching Worker worker2
Watcher: Watching Worker worker3
Watcher: Watching Worker worker4
Worker: Computing Range 1 3750
Worker: Computing Range 3751 7500
Worker: Computing Range 7501 11250
Worker: Computing Range 11251 15000
WorkerChaos: Killing worker1
Watcher: Watching Worker worker1
Worker: Computing Range 1 3750
WorkerChaos: Killing worker4
Watcher: Watching Worker worker4
Worker: Computing Range 11251 15000
WorkerChaos: Killing worker2
Watcher: Watching Worker worker2
[true,true,true]
Worker: Computing Range 3751 7500
Server: Received Sum 4275174
Server: Received Sum 12824238
Server: Received Sum 21371600
Server: Received Sum 29923304
Server: Sum of totients: 68394316
Server: Time taken in Secs, MicroSecs 8 320817
10>
```

In this section of the report include:
+ the output from execution of your program with `testRobust(4,3)`, `testRobust(8,6)`, `testRobust(12,10)`
+ a brief paragraph discussing any interesting features of the program (optional);
+ the source of your program (as Appendix E).

Section 5 Comparative Parallel Performance Measurements (12 marks)

You should measure and record the following results in numbered sections of your report. Runtime measurements should be the middle (median) value of three executions on a GPG cluster node.

N.B.1. To make a fair comparison you should measure the `totientrangeNWorkers` Erlang program, as neither C+OpenMP nor Go program has any reliability mechanisms.

N.B.2. For comparison purposes the performance of the C+OpenMP, Erlang and Go systems must be reported on the *same* graph. You may also plot other graphs to show interesting features, or use larger numbers of cores/threads.

Section 5.1 Runtimes.

DS1: execution times for the sequential C, Erlang, and Go programs, and the parallel C+OpenMP, Erlang and Go programs on 1, 2, 4, 8, 12, 16, 24, 32, 48, 64 threads/goroutines/`totientWorker` processes on a GPG Cluster node.

DS2: execution times for the sequential C, Erlang, and Go programs, and the parallel C+OpenMP, Erlang and Go programs on 1, 2, 4, 8, 12, 16, 24, 32, 48, 64 threads/goroutines/`totientWorker` processes on a GPG Cluster node.

DS3: execution times for the parallel C+OpenMP, Erlang and Go programs on 8, 16, 32, 64, threads/ goroutines on a GPG Cluster node.

Section 5.2 Speedups.

Plot three absolute speedup graphs corresponding to the runtime results for DS1, DS2 and DS3 showing the ideal speedup and the speedups for the C+OpenMP, Erlang, and Go programs. Recall that absolute speedup is calculated using the runtime of the sequential program on a single core.

Section 5.3

Complete the table below summarising the sequential performance and the best parallel runtimes of your C+OpenMP and Go programs.

Language	Sequential Runtime (s)	Best Parallel Runtime (s)	Best Speedup	No. Cores
C+OpenMP				
Erlang				
Go				

Table 2: Comparing C+OpenMP, Erlang, and Go Parallel Runtimes and Speedups

Section 5.4

A discussion of the comparative performance of the C+OpenMP, Erlang, and Go programs. **Max 1 A4 page.**

Section 6 Parallel Programming Model Comparison (6 marks).

An evaluation of the Go and C+OpenMP parallel programming models for the totient application. You should indicate any challenges you encountered in constructing your programs and the situations where each technology may usefully be applied. The comparison should be based on the TotientRange application, focus on the technology in general, and be supported by technical arguments. **Max 1 A4 page.**

Section 7 Reflection on Parallel Programming Models MSc Students Only (10 marks)

An evaluation of the C+OpenMP and Go parallel programming models in general. This section should discuss the advantages and disadvantages of the programming models. It should discuss issues related to the parallel performance, programmability and usability of the models. It should address issues of portability, in principle, discussing which kind of architectures are best targeted with each programming model. This discussion needs to be general, but can draw on the experience you gained in using the models on the Totient application. **Max 1 A4 page.**

Appendix A: C+OpenMP TotientRange Program (5 marks)

A listing of your C+OpenMP TotientRange program, clearly labelled with the author's name. Also include a paragraph, and possibly diagram(s), identifying the parallel paradigm used, and performance tuning approaches used.

Appendix B: Go TotientRange Program (5 marks)

A listing of your Go TotientRange program, clearly labelled with the author's name. Also include a paragraph, and possibly diagram(s), identifying the parallel paradigm used and performance tuning approaches used.

Appendix C: Erlang `totientrange2Workers` **Program** (5 marks)

A listing of your Erlang `totientrange2Workers` program, clearly labelled with the author's name.

Appendix D: Erlang `totientrangeNWorkers` **Program** (5 marks)

A listing of your Erlang `totientrangeNWorkers` program, clearly labelled with the author's name.

Appendix E: Erlang `totientrangeNWorkersReliable` **Program** (5 marks)

A listing of your Erlang `totientrangeNWorkersReliable` program, clearly labelled with the author's name.