# Contents

# 1 Overview

The codebase `game-of-life`, which can be found at `https://github.com/dgsaf/game-of-life`, consists of the original code provided by Dr Pascal Elahi, with the following additions:

- `src/02_gol_cpu_serial_fort.f90`: a serial GOL code which derives from `src/01_gol_cpu_serial_fort.f90`, but improves loop ordering to match the column-major format of Fortran.
- `src/02_gol_cpu_openmp_loop_fort.f90`: a parallel GOL code which derives from `src/02_gol_cpu_serial_fort.f90`, but implements OMP parallel do loops to yield performance benefits.
- `profiling/`: a directory which includes the profiling results of the `01_gol_cpu_serial_fort` and `02_gol_cpu_serial_fort` versions of the GOL code, as well as a brief summary and comparison of the profiling results. The results were collected for a GOL simulation on a $1000 \times 1000$ grid, for 100 steps with no visualisation enabled.
- `gol-job-submission.slurm`: a bash script which submits a `sbatch` job request for a GOL simulation with a given set of parameters which include:
  - `version_name`
  - `n_omp`
  - `grid_height`
  - `grid_width`
  - `num_steps`
  - `intial_conditions_type`
  - `visualisation_type`
  - `rule_type`
  - `neighbour_type`
  - `boundary_type`

  An output directory is created for the given set of parameters, with the logging output and statistics of the GOL simulation confined there. If the output directory already exists, the job isn't submitted to prevent repeating work needlessly.
- `gol-job-set-submission.sh`: a bash script which constructs different sets of parameters, and executes `gol-job-submission.slurm` for each parameter set. The batches of jobs submitted are:
  - A verification batch, which submits a job for every version of the GOL simulation, on a $10 \times 10$ grid, for 10 steps, with ASCII visualisation. This is intended to allow for visual confirmation that each version produces uniform results. The logging output and statistics are compared to verify this.

  - A scaling batch, which submits a job for every version of the GOL simulation, on a range of grid sizes, $2^n \times 2^n$ for $n = 1, \ldots, 14$, for 100 steps, with no visualisation. This is intended to collect data for analysing the scaling behaviour of each version with increasing grid size, with the total elapsed time being compared.
  - An OMP batch, which submits a job for every parallel version of the GOL simulation, for a range of assigned threads, $n_{omp} = 1, \ldots, 16$, on a $10 \times 10$ grid, for 10 steps, with ASCII visualisation. This is intended to allow for visual confirmation that each parallel version produces uniform results, independent of the number of threads assigned to the program. The logging output and statistics are compared to verify this.
  - An OMP batch, which submits a job for every parallel version of the GOL simulation, for a range of assigned threads, $n_{omp} = 1, \ldots, 16$, and for a range of grid sizes, $2^n \times 2^n$ for $n = 1, \ldots 14$ with no visualisation. This is intended to collect data for analysing the scaling behaviour of each parallel version, for each number of threads, with the total elapsed time being compared. Analysing this will yield some insight into the cost associated with the overhead of OMP threading. The total elapsed times are compared to investigate this.

- `output/`: a directory which includes subdirectories (for each parameter set submitted), each of which include the logging output and statistics file; that is,
  `output/<unique_parameter_set>/log.txt` and
  `output/<unique_parameter_set>/stats.txt`.

- `report/`: a directory which includes this `.tex` file and other files suitable for submission of this assignment.

Additionally, some minor modifications have been made to the following files:

- `Makefile`: the make rule `make cpu_serial_fort` has been modified to include `src/02_gol_cpu_serial_fort.f90`.

- `src/common_fort.f90`: the length of the variable `arg` has been increased from 32 to 2000 to allow for larger filenames for the variable `statsfile`.

- `src/01_gol_cpu_serial_fort.f90`: a bug in the `game_of_life_stats()` subroutine has been

# 2 Code Synopsis

Here, we will provide a synopsis of the most significant changes and additions to the codebase. Small, but important, sections of code will be presented and discussed; the codebase in its entirety is presented in Appendix A.

## 2.1 Bash Scripts

## 2.2 Serial Code

```
126     integer, dimension(0:NUMSTATES-1) :: num_in_state
127     real*4, dimension(0:NUMSTATES-1) :: frac
```

```
150     do i = 0, NUMSTATES-1
151         write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
152     end do
```

### 2.2.1 Original Serial Code

### 2.2.2 Improved Serial Code

### 2.2.3 Profiling Comparison

## 2.3 Parallel Loop Code

# 3 Results

## 3.1 Uniform Behaviour Verification

## 3.2 Scaling Behaviour

## 3.3 Thread Independence Verification

## 3.4 Thread Scaling Behaviour

# A Appendix

## A.1 src/common_fort.f90

```
!> Conway's Game of Life - Common
!>
!> This module provides a common set of functionality which is used across all
!> serial and parallel versions of the GOL code.
!>
!> The only modification to this module has been adjusting the length of the
!> 'arg' variable in getinput() from 32 to 2000, to accommodate long statsfile
!> filenames.
module gol_common
!-----------------------------------------------------------------------
!
!  Common routines and functions for Conway's Game of Life
!
!-----------------------------------------------------------------------
    use, intrinsic :: iso_c_binding
    integer, parameter :: NUMSTATES = 4
    integer, parameter :: CellState_ALIVE = 0
    integer, parameter :: CellState_DEAD = 1
    integer, parameter :: CellState_DYING = 2
    integer, parameter :: CellState_BORN = 3

    integer, parameter :: NUMVISUAL = 4
    integer, parameter :: VisualiseType_VISUAL_ASCII = 0
    integer, parameter :: VisualiseType_VISUAL_PNG = 1
    integer, parameter :: VisualiseType_VISUAL_OPENGL = 2
    integer, parameter :: VisualiseType_VISUAL_NONE = 3

    integer, parameter :: NUMICS = 2
    integer, parameter :: ICType_IC_RAND = 0
    integer, parameter :: ICType_IC_FILE = 1

    integer, parameter :: NUMRULES = 3
    integer, parameter :: RuleType_RULE_STANDARD = 0
    integer, parameter :: RuleType_RULES_EXTENDED = 1
    integer, parameter :: RuleType_RULES_PROB = 2

    integer, parameter :: NUMNEIGHBOURCHOICES = 2
    integer, parameter :: NeighbourType_NEIGHBOUR_STANDARD = 0
    integer, parameter :: NeighbourType_NEIGHBOUR_EXTENDED = 1

    integer, parameter :: NUMBOUNDARYCHOICES = 4
    integer, parameter :: BoundaryType_BOUNDARY_HARD = 0
    integer, parameter :: BoundaryType_BOUNDARY_TORAL = 1
    integer, parameter :: BoundaryType_BOUNDARY_TORAL_X_HARD_Y = 2
    integer, parameter :: BoundaryType_BOUNDARY_TORAL_Y_HARD_X = 3

    type Options
        integer :: n, m, nsteps
        integer :: iictype
        integer :: ivisualisetype
        integer :: iruletype
        integer :: ineighbourtype
        integer :: iboundarytype
```

```
54          character(len=2000) :: statsfile
55      end type Options
56
57  contains
58
59      !   ascii visualisation
60      subroutine visualise_ascii(step, grid, n, m)
61          implicit none
62          integer, intent(in) :: step, n, m
63          integer, dimension(:,:), intent(in) :: grid
64          character :: cell
65          integer :: i, j
66
67          write(*,*) "Game of Life"
68          write(*,*) "Step ", step
69          do i = 1, n
70              do j = 1, m
71                  cell = ' '
72                  ! could use where
73                  if (grid(i,j) .eq. CellState_ALIVE) cell = '*'
74                  write(*,"(A)", advance="no") cell
75              end do
76              write(*,*) ""
77          end do
78      end subroutine
79
80      ! png visualisation
81      subroutine visualise_png(step, grid, n, m)
82          implicit none
83          integer, intent(in) :: step, n, m
84          integer, dimension(:,:), intent(in) :: grid
85
86      end subroutine
87
88      ! no visualisation
89      subroutine visualise_none(step)
90          implicit none
91          integer, intent(in) :: step
92          write(*,*) "Game of Life, Step ", step
93      end subroutine
94
95      ! visualisation routine
96      subroutine visualise(ivisualisechoice, step, grid, n, m)
97          implicit none
98          integer, intent(in) :: ivisualisechoice
99          integer, intent(in) :: step, n, m
100         integer, dimension(:,:), intent(inout) :: grid
101         if (ivisualisechoice .eq. VisualiseType_VISUAL_ASCII) then
102             call visualise_ascii(step, grid, n, m)
103         else if (ivisualisechoice .eq. VisualiseType_VISUAL_PNG) then
104             call visualise_png(step, grid, n, m)
105         else
106             call visualise_none(step)
107         end if
108
109     end subroutine
110
111     ! generate random IC
```

```
112     subroutine generate_rand_IC(grid, n, m)
113         implicit none
114         integer, intent(in) :: n, m
115         integer, dimension(:,:), intent(inout) :: grid
116         real :: xrand, rand
117         integer :: i, j
118         do i = 1, n
119             do j = 1, m
120 #if defined(_CRAYFTN) || defined(_INTELFTN)
121                 call RANDOM_NUMBER(xrand)
122 #else
123                 xrand=rand()
124 #endif
125                 if (xrand .lt. 0.4) then
126                     grid(i,j) = CellState_DEAD
127                 else
128                     grid(i,j) = CellState_ALIVE
129                 end if
130             end do
131         end do
132 
133     end subroutine
134 
135     ! generate IC
136     subroutine generate_IC(ic_choice, grid, n, m)
137         implicit none
138         integer, intent(in) :: ic_choice
139         integer, intent(in) :: n, m
140         integer, dimension(:,:), intent(inout) :: grid
141         if (ic_choice .eq. ICType_IC_RAND) then
142             call generate_rand_IC(grid, n, m)
143         end if
144     end subroutine
145 
146     ! get some basic timing info
147     !struct timeval init_time();
148     ! get the elapsed time relative to start, return current wall time
149     !struct timeval get_elapsed_time(struct timeval start);
150 
151     ! UI
152     subroutine getinput(opt)
153         implicit none
154         type(Options), intent(inout) :: opt
155         character(len=2000) :: cmd
156         character(len=2000) :: arg
157         character(len=2000) :: statsfilename
158         integer :: count
159         integer*8 :: nbytes
160         real*4 :: memfootprint
161         ! get the commands passed and the number of args passed
162         call get_command(cmd)
163         count = command_argument_count()
164         if (count .lt. 2) then
165             write(*,*) "Usage: <grid height> <grid width> "
166             write(*,*) "[<nsteps> <IC type> <Visualisation type> <Rule type> <
    Neighbour type> "
167             write(*,*) "<Boundary type> <stats filename> ]"
168             call exit();
```

```
169         end if
170
171         statsfilename = "GOL-stats.txt"
172         call get_command_argument(1,arg)
173         read(arg,*) opt%n
174         call get_command_argument(2,arg)
175         read(arg,*) opt%m
176         opt%nsteps = -1
177         opt%ivisualisetype = VisualiseType_VISUAL_ASCII
178         opt%iruletype = RuleType_RULE_STANDARD
179         opt%iictype = ICType_IC_RAND
180         opt%ineighbourtype = NeighbourType_NEIGHBOUR_STANDARD
181         opt%iboundarytype = BoundaryType_BOUNDARY_HARD
182         if (count .ge. 3) then
183             call get_command_argument(3,arg)
184             read(arg,*) opt%nsteps
185         end if
186         if (count .ge. 4) then
187             call get_command_argument(4,arg)
188             read(arg,*) opt%iictype
189         end if
190         if (count .ge. 5) then
191             call get_command_argument(5,arg)
192             read(arg,*) opt%ivisualisetype
193         end if
194         if (count .ge. 6) then
195             call get_command_argument(6,arg)
196             read(arg,*) opt%iruletype
197         end if
198         if (count .ge. 7) then
199             call get_command_argument(7,arg)
200             read(arg,*) opt%ineighbourtype
201         end if
202         if (count .ge. 8) then
203             call get_command_argument(8,arg)
204             read(arg,*) opt%iboundarytype
205         end if
206         if (count .ge. 9) then
207             call get_command_argument(9,arg)
208             read(arg,*) statsfilename
209         end if
210         if (opt%n .le. 0 .or. opt%m .le. 0) then
211             write(*,*) "Invalid grid size."
212             call exit(1)
213         end if
214         opt%statsfile = statsfilename
215         nbytes = sizeof(opt%n) * opt%n * opt%m
216         memfootprint = real(nbytes)/1024.0/1024.0/1024.0
217         write(*,*) "Requesting grid size of ", opt%n, opt%m
218         write(*,*) " which requires", memfootprint, " GB "
219 #ifndef USEPNG
220         if (opt%ivisualisetype .eq. VisualiseType_VISUAL_PNG) then
221             write(*, *) "PNG visualisation not enabled at compile time,"
222             write(*, *) "turning off visualisation from now on."
223         end if
224 #endif
225     end subroutine
226
```

```
227     ! get some basic timing info
228     real*8 function init_time()
229         integer, dimension(8) :: value
230         call date_and_time(VALUES=value)
231         init_time = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
232         return
233     end function
234     ! get the elapsed time relative to start
235     subroutine get_elapsed_time(start)
236         real*8, intent(in) :: start
237         real*8 :: finish, delta
238         integer, dimension(8) :: value
239         call date_and_time(VALUES=value)
240         finish = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
241         delta = finish - start
242         write(*,*) "Elapsed time is ", delta, "s"
243     end subroutine
244
245 end module
```

## A.2 src/01_gol_cpu_serial_fort.f90

```
1
2  program GameOfLife
3  !-----------------------------------------------------------------------
4  !
5  !  This program runs Conway's Game of Life
6  !
7  !  Uses:
8  !
9  !-----------------------------------------------------------------------
10     use gol_common
11     implicit none
12     interface
13     ! GOL prototypes
14         subroutine game_of_life(opt, current_grid, next_grid, n, m)
15             use gol_common
16             implicit none
17             type(Options), intent(in) :: opt
18             integer, intent(in) :: n, m
19             integer, dimension(:,:), intent(in) :: current_grid
20             integer, dimension(:,:), intent(out) :: next_grid
21         end subroutine
22         ! GOL stats protoype
23         subroutine game_of_life_stats(opt, steps, current_grid)
24             use gol_common
25             implicit none
26             type(Options), intent(in) :: opt
27             integer, intent(in) :: steps
28             integer, dimension(:,:), intent(in) :: current_grid
29         end subroutine
30     end interface
31     type(Options) :: opt
32     integer :: n, m, nsteps, current_step
33     integer, dimension(:,:), allocatable :: grid, updated_grid
34     real*8 :: time1, time2
35
36     call getinput(opt)
37     n = opt%n
38     m = opt%m
39     nsteps = opt%nsteps
40     write(*,*) n, m, nsteps
41     allocate(grid(n,m))
42     allocate(updated_grid(n,m))
43     call generate_IC(opt%iictype, grid, n, m)
44     time1 = init_time()
45     current_step = 0
46     do while (current_step .ne. nsteps)
47         time2 = init_time()
48         call visualise(opt%ivisualisetype, current_step, grid, n, m);
49         call game_of_life_stats(opt, current_step, grid);
50         call game_of_life(opt, grid, updated_grid, n, m);
51         ! update current grid
52         grid(:,:) = updated_grid(:,:)
53         current_step = current_step + 1
54         call get_elapsed_time(time2)
55         time2 = init_time()
56     end do
```

```
57      write(*,*) "Finnished GOL"
58      call get_elapsed_time(time1);
59      deallocate(grid)
60      deallocate(updated_grid)
61  end program GameOfLife
62
63  ! GOL
64  subroutine game_of_life(opt, current_grid, next_grid, n, m)
65      use gol_common
66      implicit none
67      type(Options), intent(in) :: opt
68      integer, intent(in) :: n, m
69      integer, dimension(:,:), intent(in) :: current_grid
70      integer, dimension(:,:), intent(out) :: next_grid
71      integer :: neighbours, i, j, k
72      integer, dimension(8) :: n_i, n_j
73
74      ! loop over current grid and determine next grid
75      do i = 1, n
76          do j = 1, m
77              ! count the number of neighbours, clockwise around the current cell.
78              neighbours = 0;
79              n_i(1) = i - 1
80              n_j(1) = j - 1
81              n_i(2) = i - 1
82              n_j(2) = j
83              n_i(3) = i - 1
84              n_j(3) = j + 1
85              n_i(4) = i
86              n_j(4) = j + 1
87              n_i(5) = i + 1
88              n_j(5) = j + 1
89              n_i(6) = i + 1
90              n_j(6) = j
91              n_i(7) = i + 1
92              n_j(7) = j - 1
93              n_i(8) = i
94              n_j(8) = j - 1
95
96              ! loop over all neighbours and check there state
97              do k = 1, 8
98                  if(n_i(k) .ge. 1 .and. n_j(k) .ge. 1 .and. n_i(k) .le. n .and. n_j(k
    ) .le. m) then
99                      if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
100                         neighbours = neighbours + 1
101                     end if
102                 end if
103             end do
104
105             ! set the next grid
106             if(current_grid(i,j) .eq. CellState_ALIVE .and. (neighbours .eq. 2 .or.
    neighbours .eq. 3)) then
107                 next_grid(i,j) = CellState_ALIVE
108             else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3)
    then
109                 next_grid(i,j) = CellState_ALIVE
110             else
111                 next_grid(i,j) = CellState_DEAD
```

```
112             end if
113         end do
114     end do
115 end subroutine
116 
117 ! GOL stats
118 subroutine game_of_life_stats(opt, step, current_grid)
119     use gol_common
120     implicit none
121     type(Options), intent(in) :: opt
122     integer, intent(in) :: step
123     integer, dimension(:,:), intent(in) :: current_grid
124     integer :: i, j, state
125     integer*8 :: ntot
126     integer, dimension(0:NUMSTATES-1) :: num_in_state
127     real*4, dimension(0:NUMSTATES-1) :: frac
128     character(len=30) :: fmt
129 
130     fmt = "(A15,I1,A3,F10.4,A4)"
131     ntot = opt%n * opt%m
132     num_in_state = 0;
133     do i = 1, opt%n
134         do j = 1, opt%m
135             state = current_grid(i,j)
136             num_in_state(state) = num_in_state(state) + 1
137         end do
138     end do
139     frac = num_in_state/real(ntot)
140     if (step .eq. 0) then
141         open(10, file=opt%statsfile, access="sequential")
142     else
143 #if defined(_CRAYFTN) || defined(_INTELFTN)
144         open(10, file=opt%statsfile, position="append")
145 #else
146         open(10, file=opt%statsfile, access="append")
147 #endif
148     end if
149     write(10,*) "step ", step
150     do i = 0, NUMSTATES-1
151         write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
152     end do
153     write(10,*) ""
154     close(10)
155 end subroutine
```

### A.3 src/02_gol_cpu_serial_fort.f90

```
!> Conway's Game of Life.
!>
!> A cellular automata program which utilises:
!> - a 2D grid with hard, torodial or a hybrid boundary,
!> - possible cell states S = {0, 1}; that is, dead or alive,
!> - a Moore neighbourhood; that is, where for a given cell, the cells directly
!>   and diagonally adjacent are considered its neighbours,
!> - Conway's update rule, which updates the state of a cell in accordance with
!>   the following behaviour:
!>   - any live cell with two or three live neighbours continues to live,
!>   - any dead cell with three live neighbours becomes a live cell,
!>   - all other live cells die, and all other dead cells stay dead.
!>
!> This version of the program, 02_gol_cpu_serial_fort, is a serial code, which
!> is only modified from the original code, 01_gol_cpu_serial_fort, in the
!> following ways:
!> - Inner and outer loops over grid(i, j) have been swapped to ensure more
!>   efficient array caching (noting that Fortran is column-major).
!> - Fixed error in game_of_life_stats() which was indexing states from
!>   [1 .. numstates] rather than [0 .. numstates-1].
!> - Cosmetic changes, such as adding white space, and including more detailed
!>   comments.
program GameOfLife
  use gol_common
  implicit none

  interface
    ! GOL prototypes.
    subroutine game_of_life(opt, current_grid, next_grid, n, m)
      use gol_common
      implicit none

      type(Options), intent(in) :: opt
      integer, intent(in) :: n, m
      integer, dimension(:,:), intent(in) :: current_grid
      integer, dimension(:,:), intent(out) :: next_grid
    end subroutine game_of_life

    ! GOL stats protoype.
    subroutine game_of_life_stats(opt, steps, current_grid)
      use gol_common
      implicit none

      type(Options), intent(in) :: opt
      integer, intent(in) :: steps
      integer, dimension(:,:), intent(in) :: current_grid
    end subroutine game_of_life_stats
  end interface

  ! GOL main loop variables.
  type(Options) :: opt
  integer :: n, m, nsteps, current_step
  integer, dimension(:,:), allocatable :: grid, updated_grid
  real*8 :: time1, time2

  ! GOL initialisation.
```

```
57    call getinput(opt)
58
59    n = opt%n
60    m = opt%m
61    nsteps = opt%nsteps
62
63    write(*,*) n, m, nsteps
64
65    allocate(grid(n,m))
66    allocate(updated_grid(n,m))
67
68    call generate_IC(opt%iictype, grid, n, m)
69
70    ! GOL main loop.
71    time1 = init_time()
72    current_step = 0
73
74    do while (current_step .ne. nsteps)
75      time2 = init_time()
76
77      ! Visualise the current state of the grid according to the visualisation
78      ! type selected.
79      call visualise(opt%ivisualisetype, current_step, grid, n, m);
80
81      ! Calculate the statistics of the current state of the grid; that is, the
82      ! fractional occupation of each state across all cells.
83      call game_of_life_stats(opt, current_step, grid);
84
85      ! Calculate the next state of grid according to the Conway update rule.
86      call game_of_life(opt, grid, updated_grid, n, m);
87
88      ! Update the current grid variable.
89      grid(:,:) = updated_grid(:,:)
90
91      current_step = current_step + 1
92
93      ! Write out the time taken for this loop.
94      call get_elapsed_time(time2)
95
96      time2 = init_time()
97    end do
98
99    write(*,*) "Finished GOL"
100
101   ! Write out the time taken for the entire program
102   call get_elapsed_time(time1);
103
104   ! GOL cleanup.
105   deallocate(grid)
106   deallocate(updated_grid)
107 end program GameOfLife
108
109 !> GOL
110 subroutine game_of_life(opt, current_grid, next_grid, n, m)
111   use gol_common
112   implicit none
113
114   type(Options), intent(in) :: opt
```

```
115   integer, intent(in) :: n, m
116   integer, dimension(:,:), intent(in) :: current_grid
117   integer, dimension(:,:), intent(out) :: next_grid
118   integer :: neighbours, i, j, k
119   integer, dimension(8) :: n_i, n_j
120
121   ! Loop over current grid and determine next grid.
122   ! Inner and outer loops have been swapped due to Fortran storing arrays in
123   ! column-major order.
124   do j = 1, m
125     do i = 1, n
126       ! Count the number of neighbours, clockwise around the current cell.
127       neighbours = 0;
128
129       n_i(1) = i - 1
130       n_j(1) = j - 1
131
132       n_i(2) = i - 1
133       n_j(2) = j
134
135       n_i(3) = i - 1
136       n_j(3) = j + 1
137
138       n_i(4) = i
139       n_j(4) = j + 1
140
141       n_i(5) = i + 1
142       n_j(5) = j + 1
143
144       n_i(6) = i + 1
145       n_j(6) = j
146
147       n_i(7) = i + 1
148       n_j(7) = j - 1
149
150       n_i(8) = i
151       n_j(8) = j - 1
152
153       ! Loop over all neighbours and check their state. The total number of live
154       ! neighbours is accumulated.
155       do k = 1, 8
156         if(n_i(k) .ge. 1 .and. n_j(k) .ge. 1 .and. &
157             n_i(k) .le. n .and. n_j(k) .le. m) then
158           if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
159             neighbours = neighbours + 1
160           end if
161         end if
162       end do
163
164       ! Set the next grid, according to Conway's update rule.
165       if(current_grid(i,j) .eq. CellState_ALIVE .and. &
166           (neighbours .eq. 2 .or. neighbours .eq. 3)) then
167         next_grid(i,j) = CellState_ALIVE
168       else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3) then
169         next_grid(i,j) = CellState_ALIVE
170       else
171         next_grid(i,j) = CellState_DEAD
172       end if
```

```
173       end do
174     end do
175   end subroutine game_of_life
176
177   !> GOL stats
178   subroutine game_of_life_stats(opt, step, current_grid)
179     use gol_common
180     implicit none
181
182     type(Options), intent(in) :: opt
183     integer, intent(in) :: step
184     integer, dimension(:,:), intent(in) :: current_grid
185     integer :: i, j, state
186     integer*8 :: ntot
187     integer, dimension(0:NUMSTATES-1) :: num_in_state
188     real*8, dimension(0:NUMSTATES-1) :: frac
189     character(len=30) :: fmt
190
191     fmt = "(A15,I1,A3,F10.4,A4)"
192     ntot = opt%n * opt%m
193
194     ! Calculated the number of cells in each state across the entire grid.
195     ! Inner and outer loops have been swapped due to Fortran storing arrays in
196     ! column-major order.
197     num_in_state(:) = 0;
198     do j = 1, opt%m
199       do i = 1, opt%n
200         state = current_grid(i,j)
201         num_in_state(state) = num_in_state(state) + 1
202       end do
203     end do
204
205     ! Converted the state occupation from absolute terms to fractional terms.
206     frac(:) = num_in_state(:)/real(ntot)
207
208     if (step .eq. 0) then
209       open(10, file=opt%statsfile, access="sequential")
210     else
211   #if defined(_CRAYFTN) || defined(_INTELFTN)
212       open(10, file=opt%statsfile, position="append")
213   #else
214       open(10, file=opt%statsfile, access="append")
215   #endif
216     end if
217
218     write(10,*) "step ", step
219
220     do i = 0, NUMSTATES-1
221       write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
222     end do
223
224     write(10,*) ""
225     close(10)
226   end subroutine game_of_life_stats
```

## A.4 src/02_gol_cpu_openmp_loop_fort.f90

```
!> Conway's Game of Life.
!>
!> A cellular automata program which utilises:
!> - a 2D grid with hard, torodial or a hybrid boundary,
!> - possible cell states S = {0, 1}; that is, dead or alive,
!> - a Moore neighbourhood; that is, where for a given cell, the cells directly
!>   and diagonally adjacent are considered its neighbours,
!> - Conway's update rule, which updates the state of a cell in accordance with
!>   the following behaviour:
!>   - any live cell with two or three live neighbours continues to live,
!>   - any dead cell with three live neighbours becomes a live cell,
!>   - all other live cells die, and all other dead cells stay dead.
!>
!> This version of the program, 02_gol_cpu_openmp_loop_fort, is a parallel code,
!> modified from 02_gol_cpu_serial_fort, to utilise OMP loop parallelisation.
!> The following loops have been parallelised
!> - In game_of_life(..), over the (i, j) indexes as the updating of each cell's
!>   state can be performed independently of any other state.
!> - In game_of_life_stats(..), over the (i, j) indexes in calculating the
!>   number of cells in a given state.
program GameOfLife

  use omp_lib
  use gol_common
  implicit none

  interface
    ! GOL prototypes.
    subroutine game_of_life(opt, current_grid, next_grid, n, m)
      use gol_common
      implicit none

      type(Options), intent(in) :: opt
      integer, intent(in) :: n, m
      integer, dimension(:,:), intent(in) :: current_grid
      integer, dimension(:,:), intent(out) :: next_grid
    end subroutine game_of_life

    ! GOL stats protoype.
    subroutine game_of_life_stats(opt, steps, current_grid)
      use gol_common
      implicit none

      type(Options), intent(in) :: opt
      integer, intent(in) :: steps
      integer, dimension(:,:), intent(in) :: current_grid
    end subroutine game_of_life_stats
  end interface

  ! GOL main loop variables.
  type(Options) :: opt
  integer :: n, m, nsteps, current_step
  integer, dimension(:,:), allocatable :: grid, updated_grid
  real*8 :: time1, time2

  ! GOL initialisation.
```

```
57    call getinput(opt)
58
59    n = opt%n
60    m = opt%m
61    nsteps = opt%nsteps
62
63    write(*,*) n, m, nsteps
64
65    allocate(grid(n,m))
66    allocate(updated_grid(n,m))
67
68    call generate_IC(opt%iictype, grid, n, m)
69
70    ! GOL main loop.
71    time1 = init_time()
72    current_step = 0
73
74    do while (current_step .ne. nsteps)
75      time2 = init_time()
76
77      ! Visualise the current state of the grid according to the visualisation
78      ! type selected.
79      call visualise(opt%ivisualisetype, current_step, grid, n, m);
80
81      ! Calculate the statistics of the current state of the grid; that is, the
82      ! fractional occupation of each state across all cells.
83      call game_of_life_stats(opt, current_step, grid);
84
85      ! Calculate the next state of grid according to the Conway update rule.
86      call game_of_life(opt, grid, updated_grid, n, m);
87
88      ! Update the current grid variable.
89      grid(:,:) = updated_grid(:,:)
90
91      current_step = current_step + 1
92
93      ! Write out the time taken for this loop.
94      call get_elapsed_time(time2)
95
96      time2 = init_time()
97    end do
98
99    write(*,*) "Finished GOL"
100
101   ! Write out the time taken for the entire program
102   call get_elapsed_time(time1);
103
104   ! GOL cleanup.
105   deallocate(grid)
106   deallocate(updated_grid)
107 end program GameOfLife
108
109 !> GOL
110 subroutine game_of_life(opt, current_grid, next_grid, n, m)
111   use gol_common
112   implicit none
113
114   type(Options), intent(in) :: opt
```

```
115   integer, intent(in) :: n, m
116   integer, dimension(:,:), intent(in) :: current_grid
117   integer, dimension(:,:), intent(out) :: next_grid
118   integer :: neighbours, i, j, k
119   integer, dimension(8) :: n_i, n_j
120
121   ! Loop over current grid and determine next grid.
122   ! Inner and outer loops have been swapped due to Fortran storing arrays in
123   ! column-major order.
124   !
125   ! OMP parallel do
126   ! - A static scheduler is chosen as the expected work per iteration should be
127   !   approximately constant. Thus load balancing is unlikely to be an issue,
128   !   while minimising overhead is of particular concern - making a static
129   !   scheduler the optimal choice.
130   ! - The nested loops are not collapsed as testing revealed that including a
131   !   'collapse(2)' clause leads to a ~10% slower execution time.
132   !
133   !$omp parallel do &
134   !$omp   schedule (static) &
135   !$omp   shared (n, m, current_grid, next_grid) &
136   !$omp   private (i, j, n_i, n_j, k, neighbours)
137   do j = 1, m
138     do i = 1, n
139       ! Count the number of neighbours, clockwise around the current cell.
140       neighbours = 0;
141
142       n_i(1) = i - 1
143       n_j(1) = j - 1
144
145       n_i(2) = i - 1
146       n_j(2) = j
147
148       n_i(3) = i - 1
149       n_j(3) = j + 1
150
151       n_i(4) = i
152       n_j(4) = j + 1
153
154       n_i(5) = i + 1
155       n_j(5) = j + 1
156
157       n_i(6) = i + 1
158       n_j(6) = j
159
160       n_i(7) = i + 1
161       n_j(7) = j - 1
162
163       n_i(8) = i
164       n_j(8) = j - 1
165
166       ! Loop over all neighbours and check their state. The total number of live
167       ! neighbours is accumulated.
168       do k = 1, 8
169         if(n_i(k) .ge. 1 .and. n_j(k) .ge. 1 .and. &
170             n_i(k) .le. n .and. n_j(k) .le. m) then
171           if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
172             neighbours = neighbours + 1
```

```
173           end if
174         end if
175       end do
176
177       ! Set the next grid, according to Conway's update rule.
178       if(current_grid(i,j) .eq. CellState_ALIVE .and. &
179           (neighbours .eq. 2 .or. neighbours .eq. 3)) then
180         next_grid(i,j) = CellState_ALIVE
181       else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3) then
182         next_grid(i,j) = CellState_ALIVE
183       else
184         next_grid(i,j) = CellState_DEAD
185       end if
186     end do
187   end do
188   !$omp end parallel do
189 end subroutine game_of_life
190
191 !> GOL stats
192 subroutine game_of_life_stats(opt, step, current_grid)
193   use gol_common
194   implicit none
195
196   type(Options), intent(in) :: opt
197   integer, intent(in) :: step
198   integer, dimension(:,:), intent(in) :: current_grid
199   integer :: i, j, state
200   integer*8 :: ntot
201   integer, dimension(0:NUMSTATES-1) :: num_in_state
202   real*8, dimension(0:NUMSTATES-1) :: frac
203   character(len=30) :: fmt
204
205   fmt = "(A15,I1,A3,F10.4,A4)"
206   ntot = opt%n * opt%m
207
208   num_in_state(:) = 0;
209
210   ! Calculated the number of cells in each state across the entire grid.
211   ! Inner and outer loops have been swapped due to Fortran storing arrays in
212   ! column-major order.
213   !
214   ! OMP parallel do
215   ! - A static scheduler is chosen as the expected work per iteration should be
216   !   approximately constant. Thus load balancing is unlikely to be an issue,
217   !   while minimising overhead is of particular concern - making a static
218   !   scheduler the optimal choice.
219   ! - The nested loops are not collapsed as testing revealed that including a
220   !   'collapse(2)' clause leads to a ~10% slower execution time.
221   ! - A reduction clause is included for the num_in_state(:) variable as it can
222   !   be reduced across all iterations. Without this clause, the parallel do is
223   !   actually slower than the non-parallel do, however including it leads to
224   !   performance benefits.
225   !
226   !$omp parallel do &
227   !$omp   schedule (static) &
228   !$omp   shared (opt, current_grid) &
229   !$omp   private (i, j, state) &
230   !$omp   reduction(+:num_in_state)
```

```
231   do j = 1, opt%m
232     do i = 1, opt%n
233       state = current_grid(i,j)
234       num_in_state(state) = num_in_state(state) + 1
235     end do
236   end do
237   !$omp end parallel do
238
239   ! Converted the state occupation from absolute terms to fractional terms.
240   frac(:) = num_in_state(:)/real(ntot)
241
242   if (step .eq. 0) then
243     open(10, file=opt%statsfile, access="sequential")
244   else
245 #if defined(_CRAYFTN) || defined(_INTELFTN)
246     open(10, file=opt%statsfile, position="append")
247 #else
248     open(10, file=opt%statsfile, access="append")
249 #endif
250   end if
251
252   write(10,*) "step ", step
253
254   do i = 0, NUMSTATES-1
255     write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
256   end do
257
258   write(10,*) ""
259   close(10)
260 end subroutine game_of_life_stats
```

## A.5 gol-job-submission.slurm

```
#!/bin/bash -l

# SLURM details
#SBATCH --account=courses0100
#SBATCH --reservation=courses0100
#SBATCH --job-name=GOL
#SBATCH --time=00:10:00
#SBATCH --export=NONE
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=16

# parameters
# Redundant when using gol-job-set-submission.sh to export parameter sets into
# this script.
# version_name="01_gol_cpu_serial_fort"
# n_omp=2
# grid_height=10
# grid_width=10
# num_steps=10
# initial_conditions_type=0
# visualisation_type=0
# rule_type=0
# neighbour_type=0
# boundary_type=0

# filenames
base_name="\
GOL-${version_name}.\
nomp-${n_omp}.\
ngrid-${grid_height}x${grid_width}.\
nsteps-${num_steps}.\
ic_type-${initial_conditions_type}.\
vis_type-${visualisation_type}.\
rule_type-${rule_type}.\
nghbr_type-${neighbour_type}.\
bndry_type-${boundary_type}"

output_dir="output/${base_name}"
log_filename="${output_dir}/log.txt"
stats_filename="${output_dir}/stats.txt"

# load appropriate modules
module load gcc/8.3.0

# program execution
export OMP_NUM_THREADS=${n_omp}

exe="./bin/${version_name}"

args="\
${grid_height} \
${grid_width} \
${num_steps} \
${initial_conditions_type} \
${visualisation_type} \
```

```
${rule_type} \
${neighbour_type} \
${boundary_type} \
\"${stats_filename}\""

echo "GOL SLURM job submission"
# echo "version_name: ${version_name}"
echo "base_name: ${base_name}"
# echo "log_filename: ${log_filename}"
# echo "stats_filename: ${stats_filename}"
# echo "exe: ${exe}"
# echo "args: ${args}"

# check if the GOL simulation has already been performed for these parameters,
# and if it hasn't, run the GOL simulation.
if [ -d "${output_dir}" ]; then
    echo "GOL simulation already performed for these parameters"
else
    echo "GOL output directory will be created"

    mkdir -p ${output_dir}
    touch ${stats_filename}
    touch ${log_filename}

    echo "GOL simulation will commence"

    srun -n 1 -c ${n_omp} ${exe} ${args} > ${log_filename}
fi
```

## A.6 gol-job-set-submission.sh

```
#!/bin/bash -l

# kv_string
# utility function for converting parameter associative array into an export
# string suitable for the command
# > parameter_string=$(kv_string parameters)
# > "sbatch gol-job-submission.slurm --export=${parameter_string}"
function kv_string {
    local -n array=$1

    str=""

    declare -i counter=1
    length=${#array[*]}

    for key in ${!array[*]} ; do
        kv_pair="${key}=${array[${key}]}"
        if [ ${counter} != ${length} ] ; then
            str+="${kv_pair},"
        else
            str+="${kv_pair}"
        fi
        counter+=1
    done

    echo ${str}
}

# parameter sets
version_names_serial="01_gol_cpu_serial_fort 02_gol_cpu_serial_fort"
# version_names_parallel="02_gol_cpu_openmp_task_fort 02_gol_cpu_openmp_loop_fort"
version_names_parallel="02_gol_cpu_openmp_loop_fort"
version_names="${version_names_serial} ${version_names_parallel}"

grid_lengths="2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384"
# grid_lengths="10 100 1000 10000"

n_omps="1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16"

# parameter associative array with default values
declare -A parameters

parameters[version_name]=""
parameters[n_omp]=4
parameters[grid_height]=1
parameters[grid_width]=1
parameters[num_steps]=1
parameters[initial_conditions_type]=0
parameters[visualisation_type]=3
parameters[rule_type]=0
parameters[neighbour_type]=0
parameters[boundary_type]=0

# load appropriate modules and compile code
module load gcc/8.3.0

```

```
# GOL job sets
echo "GOL SLURM job set submission"
echo

# GOL uniformity / ascii visualisation
# Performing GOL simulations on a 10x10 grid, for 10 steps, with ascii
# visualisation. Intended to verify that different GOL versions produce the same
# behaviour.
echo "GOL ascii jobs"
echo "versions: ${version_names}"
echo "ngrid: 10x10"
echo "nsteps: 10"
echo "vis_type: 0 (ascii)"
for version_name in ${version_names} ; do
    parameters[version_name]=${version_name}
    parameters[grid_height]=10
    parameters[grid_width]=10
    parameters[num_steps]=10
    parameters[visualisation_type]=0

    parameter_string=$(kv_string parameters)
    # echo "--export=${parameter_string}"
    sbatch --export=${parameter_string} gol-job-submission.slurm
done
echo

# GOL scaling
# Performing GOL simulations on increasingly large grids, 2^n x 2^n for n = 1,
# .., nmax, for 100 steps, with no visualisation. Intended to determine the
# scaling behaviour of the different GOL versions.
echo "GOL scaling jobs"
echo "versions: ${version_names}"
echo "grid lengths: ${grid_lengths}"
echo "nsteps: 100"
echo "vis_type: 3 (none)"
for version_name in ${version_names} ; do
    parameters[version_name]=${version_name}
    parameters[num_steps]=100
    parameters[visualisation_type]=3

    for grid_length in ${grid_lengths}; do
        parameters[grid_height]=${grid_length}
        parameters[grid_width]=${grid_length}

        parameter_string=$(kv_string parameters)
        # echo "--export=${parameter_string}"
        sbatch --export=${parameter_string} gol-job-submission.slurm
    done
done
echo

# GOL thread independence
# Performing parallel GOL simulations with varying numbers of omp threads, on a
# 10x10 grid, for 10 steps, with ascii visualisation. Intended to verify that
# the parallel GOL versions produce the same behaviour independent of the number
# of omp threads.
echo "GOL thread independence jobs"
echo "versions: ${version_names_parallel}"
```

```
115 echo "ngrid: 10x10"
116 echo "nsteps: 10"
117 echo "vis_type: 0 (ascii)"
118 echo "n_omps: ${n_omps}"
119 for version_name in ${version_names_parallel} ; do
120     parameters[version_name]=${version_name}
121     parameters[grid_height]=10
122     parameters[grid_width]=10
123     parameters[num_steps]=10
124     parameters[visualisation_type]=0
125 
126     for n_omp in ${n_omps}; do
127         parameters[n_omp]=${n_omp}
128 
129         parameter_string=$(kv_string parameters)
130         # echo "--export=${parameter_string}"
131         sbatch --export=${parameter_string} gol-job-submission.slurm
132     done
133 done
134 echo
135 
136 # GOL thread scaling
137 # Performing parallel GOL simulations with varying numbers of omp threads, on
138 # increasing large grids, 2^n x 2^n for n = 1, .., nmax, for 100 steps, with no
139 # visualisation. Intended to determine the scaling behaviour of the parallel GOL
140 # versions, with increasing number of omp threads, and with increasing grid
141 # sizes.
142 echo "GOL thread scaling jobs"
143 echo "versions: ${version_names_parallel}"
144 echo "grid lengths: ${grid_lengths}"
145 echo "nsteps: 100"
146 echo "vis_type: 3 (none)"
147 echo "n_omps: ${n_omps}"
148 for version_name in ${version_names_parallel} ; do
149     parameters[version_name]=${version_name}
150     parameters[num_steps]=100
151     parameters[visualisation_type]=3
152 
153     for n_omp in ${n_omps}; do
154         parameters[n_omp]=${n_omp}
155 
156         for grid_length in ${grid_lengths}; do
157             parameters[grid_height]=${grid_length}
158             parameters[grid_width]=${grid_length}
159 
160             parameter_string=$(kv_string parameters)
161             # echo "--export=${parameter_string}"
162             sbatch --export=${parameter_string} gol-job-submission.slurm
163         done
164     done
165 done
166 echo
```