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 1000×1000 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 10×10 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, $\mathrm{n_{omp}} = 1, \ldots, 16$, on a 10×10 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, $\mathrm{n_{omp}} = 1, \ldots, 16$, 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

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

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

All versions of the GOL simulation were executed on a 10×10 grid, for 10 steps, with ASCII visualisation on, and with $n_{omp} = 4$ for the parallel versions of the code. The behaviour of the different versions can be compared exactly by examining the ASCII visualisation of the grids to show that they are identical, and/or by comparing the statistics of the grids produced.
To compare the ASCII visualisations for each version, using the `diff` command to compare the `output/*ngrid-10x10*/log.txt` output for each version from that of `src/01_gol_cpu_serial_fort.f90` should indicate any differences in behaviour. If each version produces identical grids, the only differences should be timing results.

```
original="output/GOL-01_gol_cpu_serial_fort.nomp-4.ngrid-10x10.nsteps-10.\
ic_type-0.vis_type-0.rule_type-0.nghbr_type-0.bndry_type-0/log.txt"

for log in output/*ngrid-10x10*/log.txt; do
  diff ${original} ${log}
done
```

Similarly, to compare the statistics, we again use the `diff` command to compare the `output/*ngrid-10x10*/stats.txt` output for each version from that of `src/01_gol_cpu_serial_fort.f90`.

```
original="output/GOL-01_gol_cpu_serial_fort.nomp-4.ngrid-10x10.nsteps-10.\
ic_type-0.vis_type-0.rule_type-0.nghbr_type-0.bndry_type-0/stats.txt"

for stats in output/*ngrid-10x10*/stats.txt; do
  diff ${original} ${stats}
done
```

Using both methods on the data collected in `output/`, we have verified that the different versions produce identical grids and thus statistics.

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
```

```
        character(len=2000) :: statsfile
    end type Options

contains

    !   ascii visualisation
    subroutine visualise_ascii(step, grid, n, m)
        implicit none
        integer, intent(in) :: step, n, m
        integer, dimension(:,:), intent(in) :: grid
        character :: cell
        integer :: i, j

        write(*,*) "Game of Life"
        write(*,*) "Step ", step
        do i = 1, n
            do j = 1, m
                cell = ' '
                ! could use where
                if (grid(i,j) .eq. CellState_ALIVE) cell = '*'
                write(*,"(A)", advance="no") cell
            end do
            write(*,*) ""
        end do
    end subroutine

    ! png visualisation
    subroutine visualise_png(step, grid, n, m)
        implicit none
        integer, intent(in) :: step, n, m
        integer, dimension(:,:), intent(in) :: grid

    end subroutine

    ! no visualisation
    subroutine visualise_none(step)
        implicit none
        integer, intent(in) :: step
        write(*,*) "Game of Life, Step ", step
    end subroutine

    ! visualisation routine
    subroutine visualise(ivisualisechoice, step, grid, n, m)
        implicit none
        integer, intent(in) :: ivisualisechoice
        integer, intent(in) :: step, n, m
        integer, dimension(:,:), intent(inout) :: grid
        if (ivisualisechoice .eq. VisualiseType_VISUAL_ASCII) then
            call visualise_ascii(step, grid, n, m)
        else if (ivisualisechoice .eq. VisualiseType_VISUAL_PNG) then
            call visualise_png(step, grid, n, m)
        else
            call visualise_none(step)
        end if

    end subroutine

    ! generate random IC
```

```
    subroutine generate_rand_IC(grid, n, m)
        implicit none
        integer, intent(in) :: n, m
        integer, dimension(:,:), intent(inout) :: grid
        real :: xrand, rand
        integer :: i, j
        do i = 1, n
            do j = 1, m
#if defined(_CRAYFTN) || defined(_INTELFTN)
                call RANDOM_NUMBER(xrand)
#else
                xrand=rand()
#endif
                if (xrand .lt. 0.4) then
                    grid(i,j) = CellState_DEAD
                else
                    grid(i,j) = CellState_ALIVE
                end if
            end do
        end do

    end subroutine

    ! generate IC
    subroutine generate_IC(ic_choice, grid, n, m)
        implicit none
        integer, intent(in) :: ic_choice
        integer, intent(in) :: n, m
        integer, dimension(:,:), intent(inout) :: grid
        if (ic_choice .eq. ICType_IC_RAND) then
            call generate_rand_IC(grid, n, m)
        end if
    end subroutine

    ! get some basic timing info
    !struct timeval init_time();
    ! get the elapsed time relative to start, return current wall time
    !struct timeval get_elapsed_time(struct timeval start);

    ! UI
    subroutine getinput(opt)
        implicit none
        type(Options), intent(inout) :: opt
        character(len=2000) :: cmd
        character(len=2000) :: arg
        character(len=2000) :: statsfilename
        integer :: count
        integer*8 :: nbytes
        real*4 :: memfootprint
        ! get the commands passed and the number of args passed
        call get_command(cmd)
        count = command_argument_count()
        if (count .lt. 2) then
            write(*,*) "Usage: <grid height> <grid width> "
            write(*,*) "[<nsteps> <IC type> <Visualisation type> <Rule type> <Neighbour type> "
            write(*,*) "<Boundary type> <stats filename> ]"
            call exit();
```

```
        end if

        statsfilename = "GOL-stats.txt"
        call get_command_argument(1,arg)
        read(arg,*) opt%n
        call get_command_argument(2,arg)
        read(arg,*) opt%m
        opt%nsteps = -1
        opt%ivisualisetype = VisualiseType_VISUAL_ASCII
        opt%iruletype = RuleType_RULE_STANDARD
        opt%iictype = ICType_IC_RAND
        opt%ineighbourtype = NeighbourType_NEIGHBOUR_STANDARD
        opt%iboundarytype = BoundaryType_BOUNDARY_HARD
        if (count .ge. 3) then
            call get_command_argument(3,arg)
            read(arg,*) opt%nsteps
        end if
        if (count .ge. 4) then
            call get_command_argument(4,arg)
            read(arg,*) opt%iictype
        end if
        if (count .ge. 5) then
            call get_command_argument(5,arg)
            read(arg,*) opt%ivisualisetype
        end if
        if (count .ge. 6) then
            call get_command_argument(6,arg)
            read(arg,*) opt%iruletype
        end if
        if (count .ge. 7) then
            call get_command_argument(7,arg)
            read(arg,*) opt%ineighbourtype
        end if
        if (count .ge. 8) then
            call get_command_argument(8,arg)
            read(arg,*) opt%iboundarytype
        end if
        if (count .ge. 9) then
            call get_command_argument(9,arg)
            read(arg,*) statsfilename
        end if
        if (opt%n .le. 0 .or. opt%m .le. 0) then
            write(*,*) "Invalid grid size."
            call exit(1)
        end if
        opt%statsfile = statsfilename
        nbytes = sizeof(opt%n) * opt%n * opt%m
        memfootprint = real(nbytes)/1024.0/1024.0/1024.0
        write(*,*) "Requesting grid size of ", opt%n, opt%m
        write(*,*) " which requires", memfootprint, " GB "
#ifndef USEPNG
        if (opt%ivisualisetype .eq. VisualiseType_VISUAL_PNG) then
            write(*, *) "PNG visualisation not enabled at compile time,"
            write(*, *) "turning off visualisation from now on."
        end if
#endif
    end subroutine

```

```
    ! get some basic timing info
    real*8 function init_time()
        integer, dimension(8) :: value
        call date_and_time(VALUES=value)
        init_time = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
        return
    end function
    ! get the elapsed time relative to start
    subroutine get_elapsed_time(start)
        real*8, intent(in) :: start
        real*8 :: finish, delta
        integer, dimension(8) :: value
        call date_and_time(VALUES=value)
        finish = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
        delta = finish - start
        write(*,*) "Elapsed time is ", delta, "s"
    end subroutine

end module
```

A.2 src/01_gol_cpu_serial_fort.f90

```fortran

program GameOfLife
!-----------------------------------------------------------------------
!
!  This program runs Conway's Game of Life
!
!  Uses:
!
!-----------------------------------------------------------------------
    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
        ! 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
    end interface
    type(Options) :: opt
    integer :: n, m, nsteps, current_step
    integer, dimension(:,:), allocatable :: grid, updated_grid
    real*8 :: time1, time2

    call getinput(opt)
    n = opt%n
    m = opt%m
    nsteps = opt%nsteps
    write(*,*) n, m, nsteps
    allocate(grid(n,m))
    allocate(updated_grid(n,m))
    call generate_IC(opt%iictype, grid, n, m)
    time1 = init_time()
    current_step = 0
    do while (current_step .ne. nsteps)
        time2 = init_time()
        call visualise(opt%ivisualisetype, current_step, grid, n, m);
        call game_of_life_stats(opt, current_step, grid);
        call game_of_life(opt, grid, updated_grid, n, m);
        ! update current grid
        grid(:,:) = updated_grid(:,:)
        current_step = current_step + 1
        call get_elapsed_time(time2)
        time2 = init_time()
    end do
```

```
    write(*,*) "Finnished GOL"
    call get_elapsed_time(time1);
    deallocate(grid)
    deallocate(updated_grid)
end program GameOfLife

! GOL
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
    integer :: neighbours, i, j, k
    integer, dimension(8) :: n_i, n_j

    ! loop over current grid and determine next grid
    do i = 1, n
        do j = 1, m
            ! count the number of neighbours, clockwise around the current cell.
            neighbours = 0;
            n_i(1) = i - 1
            n_j(1) = j - 1
            n_i(2) = i - 1
            n_j(2) = j
            n_i(3) = i - 1
            n_j(3) = j + 1
            n_i(4) = i
            n_j(4) = j + 1
            n_i(5) = i + 1
            n_j(5) = j + 1
            n_i(6) = i + 1
            n_j(6) = j
            n_i(7) = i + 1
            n_j(7) = j - 1
            n_i(8) = i
            n_j(8) = j - 1

            ! loop over all neighbours and check there state
            do k = 1, 8
                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
                    if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
                        neighbours = neighbours + 1
                    end if
                end if
            end do

            ! set the next grid
            if(current_grid(i,j) .eq. CellState_ALIVE .and. (neighbours .eq. 2 .or. neighbours .eq. 3)) then
                next_grid(i,j) = CellState_ALIVE
            else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3) then
                next_grid(i,j) = CellState_ALIVE
            else
                next_grid(i,j) = CellState_DEAD
```

```fortran
            end if
        end do
    end do
end subroutine

! GOL stats
subroutine game_of_life_stats(opt, step, current_grid)
    use gol_common
    implicit none
    type(Options), intent(in) :: opt
    integer, intent(in) :: step
    integer, dimension(:,:), intent(in) :: current_grid
    integer :: i, j, state
    integer*8 :: ntot
    integer, dimension(0:NUMSTATES-1) :: num_in_state
    real*4, dimension(0:NUMSTATES-1) :: frac
    character(len=30) :: fmt

    fmt = "(A15,I1,A3,F10.4,A4)"
    ntot = opt%n * opt%m
    num_in_state = 0;
    do i = 1, opt%n
        do j = 1, opt%m
            state = current_grid(i,j)
            num_in_state(state) = num_in_state(state) + 1
        end do
    end do
    frac = num_in_state/real(ntot)
    if (step .eq. 0) then
        open(10, file=opt%statsfile, access="sequential")
    else
#if defined(_CRAYFTN) || defined(_INTELFTN)
        open(10, file=opt%statsfile, position="append")
#else
        open(10, file=opt%statsfile, access="append")
#endif
    end if
    write(10,*) "step ", step
    do i = 0, NUMSTATES-1
        write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
    end do
    write(10,*) ""
    close(10)
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.
```

```
  call getinput(opt)

  n = opt%n
  m = opt%m
  nsteps = opt%nsteps

  write(*,*) n, m, nsteps

  allocate(grid(n,m))
  allocate(updated_grid(n,m))

  call generate_IC(opt%iictype, grid, n, m)

  ! GOL main loop.
  time1 = init_time()
  current_step = 0

  do while (current_step .ne. nsteps)
    time2 = init_time()

    ! Visualise the current state of the grid according to the visualisation
    ! type selected.
    call visualise(opt%ivisualisetype, current_step, grid, n, m);

    ! Calculate the statistics of the current state of the grid; that is, the
    ! fractional occupation of each state across all cells.
    call game_of_life_stats(opt, current_step, grid);

    ! Calculate the next state of grid according to the Conway update rule.
    call game_of_life(opt, grid, updated_grid, n, m);

    ! Update the current grid variable.
    grid(:,:) = updated_grid(:,:)

    current_step = current_step + 1

    ! Write out the time taken for this loop.
    call get_elapsed_time(time2)

    time2 = init_time()
  end do

  write(*,*) "Finished GOL"

  ! Write out the time taken for the entire program
  call get_elapsed_time(time1);

  ! GOL cleanup.
  deallocate(grid)
  deallocate(updated_grid)
end program GameOfLife

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

  type(Options), intent(in) :: opt
```

```fortran
  integer, intent(in) :: n, m
  integer, dimension(:,:), intent(in) :: current_grid
  integer, dimension(:,:), intent(out) :: next_grid
  integer :: neighbours, i, j, k
  integer, dimension(8) :: n_i, n_j

  ! Loop over current grid and determine next grid.
  ! Inner and outer loops have been swapped due to Fortran storing arrays in
  ! column-major order.
  do j = 1, m
    do i = 1, n
      ! Count the number of neighbours, clockwise around the current cell.
      neighbours = 0;

      n_i(1) = i - 1
      n_j(1) = j - 1

      n_i(2) = i - 1
      n_j(2) = j

      n_i(3) = i - 1
      n_j(3) = j + 1

      n_i(4) = i
      n_j(4) = j + 1

      n_i(5) = i + 1
      n_j(5) = j + 1

      n_i(6) = i + 1
      n_j(6) = j

      n_i(7) = i + 1
      n_j(7) = j - 1

      n_i(8) = i
      n_j(8) = j - 1

      ! Loop over all neighbours and check their state. The total number of live
      ! neighbours is accumulated.
      do k = 1, 8
        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
          if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
            neighbours = neighbours + 1
          end if
        end if
      end do

      ! Set the next grid, according to Conway's update rule.
      if(current_grid(i,j) .eq. CellState_ALIVE .and. &
          (neighbours .eq. 2 .or. neighbours .eq. 3)) then
        next_grid(i,j) = CellState_ALIVE
      else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3) then
        next_grid(i,j) = CellState_ALIVE
      else
        next_grid(i,j) = CellState_DEAD
      end if
```

```
    end do
  end do
end subroutine game_of_life

!> GOL stats
subroutine game_of_life_stats(opt, step, current_grid)
  use gol_common
  implicit none

  type(Options), intent(in) :: opt
  integer, intent(in) :: step
  integer, dimension(:,:), intent(in) :: current_grid
  integer :: i, j, state
  integer*8 :: ntot
  integer, dimension(0:NUMSTATES-1) :: num_in_state
  real*8, dimension(0:NUMSTATES-1) :: frac
  character(len=30) :: fmt

  fmt = "(A15,I1,A3,F10.4,A4)"
  ntot = opt%n * opt%m

  ! Calculated the number of cells in each state across the entire grid.
  ! Inner and outer loops have been swapped due to Fortran storing arrays in
  ! column-major order.
  num_in_state(:) = 0;
  do j = 1, opt%m
    do i = 1, opt%n
      state = current_grid(i,j)
      num_in_state(state) = num_in_state(state) + 1
    end do
  end do

  ! Converted the state occupation from absolute terms to fractional terms.
  frac(:) = num_in_state(:)/real(ntot)

  if (step .eq. 0) then
    open(10, file=opt%statsfile, access="sequential")
  else
#if defined(_CRAYFTN) || defined(_INTELFTN)
    open(10, file=opt%statsfile, position="append")
#else
    open(10, file=opt%statsfile, access="append")
#endif
  end if

  write(10,*) "step ", step

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

  write(10,*) ""
  close(10)
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.
```

```
  call getinput(opt)

  n = opt%n
  m = opt%m
  nsteps = opt%nsteps

  write(*,*) n, m, nsteps

  allocate(grid(n,m))
  allocate(updated_grid(n,m))

  call generate_IC(opt%iictype, grid, n, m)

  ! GOL main loop.
  time1 = init_time()
  current_step = 0

  do while (current_step .ne. nsteps)
    time2 = init_time()

    ! Visualise the current state of the grid according to the visualisation
    ! type selected.
    call visualise(opt%ivisualisetype, current_step, grid, n, m);

    ! Calculate the statistics of the current state of the grid; that is, the
    ! fractional occupation of each state across all cells.
    call game_of_life_stats(opt, current_step, grid);

    ! Calculate the next state of grid according to the Conway update rule.
    call game_of_life(opt, grid, updated_grid, n, m);

    ! Update the current grid variable.
    grid(:,:) = updated_grid(:,:)

    current_step = current_step + 1

    ! Write out the time taken for this loop.
    call get_elapsed_time(time2)

    time2 = init_time()
  end do

  write(*,*) "Finished GOL"

  ! Write out the time taken for the entire program
  call get_elapsed_time(time1);

  ! GOL cleanup.
  deallocate(grid)
  deallocate(updated_grid)
end program GameOfLife

!> GOL
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
  integer :: neighbours, i, j, k
  integer, dimension(8) :: n_i, n_j

  ! Loop over current grid and determine next grid.
  ! Inner and outer loops have been swapped due to Fortran storing arrays in
  ! column-major order.
  !
  ! OMP parallel do
  ! - A static scheduler is chosen as the expected work per iteration should be
  !   approximately constant. Thus load balancing is unlikely to be an issue,
  !   while minimising overhead is of particular concern - making a static
  !   scheduler the optimal choice.
  ! - The nested loops are not collapsed as testing revealed that including a
  !   'collapse(2)' clause leads to a ~10% slower execution time.
  !
  !$omp parallel do &
  !$omp   schedule (static) &
  !$omp   shared (n, m, current_grid, next_grid) &
  !$omp   private (i, j, n_i, n_j, k, neighbours)
  do j = 1, m
    do i = 1, n
      ! Count the number of neighbours, clockwise around the current cell.
      neighbours = 0;

      n_i(1) = i - 1
      n_j(1) = j - 1

      n_i(2) = i - 1
      n_j(2) = j

      n_i(3) = i - 1
      n_j(3) = j + 1

      n_i(4) = i
      n_j(4) = j + 1

      n_i(5) = i + 1
      n_j(5) = j + 1

      n_i(6) = i + 1
      n_j(6) = j

      n_i(7) = i + 1
      n_j(7) = j - 1

      n_i(8) = i
      n_j(8) = j - 1

      ! Loop over all neighbours and check their state. The total number of live
      ! neighbours is accumulated.
      do k = 1, 8
        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
          if (current_grid(n_i(k), n_j(k)) .eq. CellState_ALIVE) then
            neighbours = neighbours + 1
```

```
          end if
        end if
      end do

      ! Set the next grid, according to Conway's update rule.
      if(current_grid(i,j) .eq. CellState_ALIVE .and. &
          (neighbours .eq. 2 .or. neighbours .eq. 3)) then
        next_grid(i,j) = CellState_ALIVE
      else if (current_grid(i,j) .eq. CellState_DEAD .and. neighbours .eq. 3) then
        next_grid(i,j) = CellState_ALIVE
      else
        next_grid(i,j) = CellState_DEAD
      end if
    end do
  end do
  !$omp end parallel do
end subroutine game_of_life

!> GOL stats
subroutine game_of_life_stats(opt, step, current_grid)
  use gol_common
  implicit none

  type(Options), intent(in) :: opt
  integer, intent(in) :: step
  integer, dimension(:,:), intent(in) :: current_grid
  integer :: i, j, state
  integer*8 :: ntot
  integer, dimension(0:NUMSTATES-1) :: num_in_state
  real*8, dimension(0:NUMSTATES-1) :: frac
  character(len=30) :: fmt

  fmt = "(A15,I1,A3,F10.4,A4)"
  ntot = opt%n * opt%m

  num_in_state(:) = 0;

  ! Calculated the number of cells in each state across the entire grid.
  ! Inner and outer loops have been swapped due to Fortran storing arrays in
  ! column-major order.
  !
  ! OMP parallel do
  ! - A static scheduler is chosen as the expected work per iteration should be
  !   approximately constant. Thus load balancing is unlikely to be an issue,
  !   while minimising overhead is of particular concern - making a static
  !   scheduler the optimal choice.
  ! - The nested loops are not collapsed as testing revealed that including a
  !   'collapse(2)' clause leads to a ~10% slower execution time.
  ! - A reduction clause is included for the num_in_state(:) variable as it can
  !   be reduced across all iterations. Without this clause, the parallel do is
  !   actually slower than the non-parallel do, however including it leads to
  !   performance benefits.
  !
  !$omp parallel do &
  !$omp   schedule (static) &
  !$omp   shared (opt, current_grid) &
  !$omp   private (i, j, state) &
  !$omp   reduction(+:num_in_state)
```

```
  do j = 1, opt%m
    do i = 1, opt%n
      state = current_grid(i,j)
      num_in_state(state) = num_in_state(state) + 1
    end do
  end do
  !$omp end parallel do

  ! Converted the state occupation from absolute terms to fractional terms.
  frac(:) = num_in_state(:)/real(ntot)

  if (step .eq. 0) then
    open(10, file=opt%statsfile, access="sequential")
  else
#if defined(_CRAYFTN) || defined(_INTELFTN)
    open(10, file=opt%statsfile, position="append")
#else
    open(10, file=opt%statsfile, access="append")
#endif
  end if

  write(10,*) "step ", step

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

  write(10,*) ""
  close(10)
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}"
```

```
echo "ngrid: 10x10"
echo "nsteps: 10"
echo "vis_type: 0 (ascii)"
echo "n_omps: ${n_omps}"
for version_name in ${version_names_parallel} ; do
    parameters[version_name]=${version_name}
    parameters[grid_height]=10
    parameters[grid_width]=10
    parameters[num_steps]=10
    parameters[visualisation_type]=0

    for n_omp in ${n_omps}; do
        parameters[n_omp]=${n_omp}

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

# GOL thread scaling
# Performing parallel GOL simulations with varying numbers of omp threads, on
# increasing 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 parallel GOL
# versions, with increasing number of omp threads, and with increasing grid
# sizes.
echo "GOL thread scaling jobs"
echo "versions: ${version_names_parallel}"
echo "grid lengths: ${grid_lengths}"
echo "nsteps: 100"
echo "vis_type: 3 (none)"
echo "n_omps: ${n_omps}"
for version_name in ${version_names_parallel} ; do
    parameters[version_name]=${version_name}
    parameters[num_steps]=100
    parameters[visualisation_type]=3

    for n_omp in ${n_omps}; do
        parameters[n_omp]=${n_omp}

        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
done
echo
```