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. Note that the sbatch job request is invoked with --cpus-per-task=16.

- 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 default values for the unspecified parameters are:

    - $n_{omp} = 4$,
    - initial_conditions_type = 0 (random initial conditions),
    - rule_type = 0 (ordinary 2D GOL rule),

- neighbour_type $= 0$ (Moore neighbourhood),
- boundary_type $= 0$ (hard boundary with default state being dead).

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, $n_{omp} = 1, \ldots, 16$, 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, $n_{omp} = 1, \ldots, 16$, 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.

- gol-data-extract.sh: a bash script which extracts the final timing data for the two scaling batches of jobs:

  - The scaling batch, which includes 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, and with no visualisation.
  - The OMP parallel thread scaling batch, which includes 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.

  A data directory is created for each scaling batch of jobs, and a .csv file is created for each unique job, if it doesn't already exist.

- data/: a directory which includes a subdirectory for each scaling batch of jobs, each of which contains a set of .csv files, consisting of the grid length and timing data, for each of the jobs in the batch.

- **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**: an error in `subroutine game_of_life_stats()` concerning array indexing has been corrected.

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 Common Code

The entire common code, `src/common_fort.f90`, can be found in subsection A.1.

Only a small modification has been made to `src/common_fort.f90` from the original code provided by Dr Pascal Elahi: the length of the variable `arg` has been increased from 32 to 2000. This is due to the slurm script `gol-job-submission.slurm` providing, as one of the command line arguments, long strings for the variable `statsfile`. This is necessary for proper manipulation of the `*/stats.txt` output file, which was being truncated prior to the change.

The modified code snippet is shown below.

```
156            ! arg length changed from 32 to 2000 to accommodate large statsfilename.
157            character(len=2000) :: arg
```

Listing 1: *Modifications to* `arg` *variable within* *subroutine* `getinput()`.

2.2 Serial Code

2.2.1 Original Serial Code

The entire original serial code, `src/01_gol_cpu_serial_fort.f90`, can be found in subsection A.2.

It contained a bug, which indexed the `num_in_state` and `frac` arrays using 1-based indexing rather than 0-based indexing (mandated by the parameters defined in `src/common_fort.f90`), resulting in incorrect statistics being written to the statistics file.

The corrected code snippets are shown below.

```
126      ! num_in_state and frac modified from dimension(NUMSTATES) to
127      ! dimension(0:NUMSTATES-1) to synchronise with the 0-based indexing of
128      ! states mandated in common_fort.f90.
129      integer, dimension(0:NUMSTATES-1) :: num_in_state
130      real*4, dimension(0:NUMSTATES-1) :: frac
```

Listing 2: *Modifications to indexing of* `num_in_state` *and* `frac` *array variable within* *subroutine* `game_of_life_stats()`.

```
153      ! Loop bounds modified to synchronise with 0-based indexing of frac(:).
154      do i = 0, NUMSTATES-1
```

5

```
155        write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
156      end do
```

Listing 3: Modifications to indexing of `frac` array variable within `subroutine` `game_of_life_stats()`.

2.2.2 Improved Serial Code

The entire improved serial code, `src/02_gol_cpu_serial_fort.f90`, can be found in .

It can be observed in `src/01_gol_cpu_serial_fort.f90`, that the nested `do` loops over the variable `grid(:, :)` do not respect Fortran's column major order for multi-dimensional arrays. Hence, it is guaranteed that cache misses are occuring as the program needs to stride over the array in memory to access elements of `grid(:, :)`. By reversing the order of these nested `do` loops, wherever they occur, the program will no longer suffer from these cache misses, and will instead collect large blocks of contiguous memory, yielding performance benefits.

The modified code snippets are shown below.

```
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
```

Listing 4: Modifications to order of nested `do` loops within `subroutine` `game_of_life()`. Mitigates problem of cache misses when accessing and updating `grid(i, j)`.

```
199    ! Calculated the number of cells in each state across the entire grid.
200    ! Inner and outer loops have been swapped due to Fortran storing arrays in
201    ! column-major order.
202    do j = 1, opt%m
203      do i = 1, opt%n
```

Listing 5: Modifications to order of nested `do` loops within `subroutine` `game_of_life_stats()`. Mitigates problem of cache misses when accessing and updating `grid(i, j)`.

2.2.3 Profiling Comparison

The two serial codes, `src/01_gol_cpu_serial_fort.f90` and `src/02_gol_cpu_serial_fort.f90` were compiled with profiling turned on, and performed a GOL simulation on a 1000×1000 grid, for 100 steps, with no visualisation. The profiling results are located in `profiling/analysis_*.txt` and a brief summary of these results, and the relevant source code snippets, are included in `profiling/profiling-summary.txt`.

The original serial code took $4.906\,$s to complete. Cumulatively $\sim 45\,\%$ of the time was spent calculating the number of living neighbours of a given cell in `subroutine` `game_of_life()`, while cumulatively $\sim 40\,\%$ of the time was spent calculating the number of grid cells that were in a given state in

subroutine `game_of_life_stats()`. That is to say, ~95 % of the time was spent accessing elements of `grid(:, :)`. This clearly shows that the vast majority of the time spent in the GOL simulation concerns accessing elements of `grid(:, :)`, and that improving the efficiency of these access operations ensures improved code performance.

The improved serial code took 2.949 s to complete. Cumulatively ~65 % of the time was spent calculating the number of living neighbours of a given cell in subroutine `game_of_life()`, while cumulatively only ~5 % of time was spent calculating the number of grid cells that were in a given state in subroutine `game_of_life_stats()`. It is clear that reversing the order of the nested `do` loops has yielded a significant performance increase, with a ~40 % reduction in time taken. Furthermore, the percentage of time spent accessing elements of `grid(:, :)` has been reduced from ~95 % to ~70 %, indicating that the access operations are less of a bottleneck.

Note that in subroutine `game_of_life()`, for every update operation performed on an element of `grid(:, :)`, 4 to 8 access operations are performed.

2.3 Parallel Loop Code

The entire parallel loop code, `src/02_gol_cpu_openmp_loop_fort.f90`, can be found in subsection A.4. It is derived from `src/02_gol_cpu_serial_fort.f90` but with modifications to incorporate parallelism.

The OpenMP worksharing construct, `parallel do`, has been introduced to parallelise the segments of the GOL simulation code which can be parallelised to yield performance improvements. From the profiling analysis, discussed in subsubsection 2.2.3, it was observed that the majority of time is spent accessing elements the variable `grid(:, :)`. Hence, the segments of code which are likely to result in the best performance improvements when parallelised, are those that involve access operations on `grid(:, :)`; namely, the nested loops over `grid(i, j)` in subroutine `game_of_life()` and subroutine `game_of_life_stats()`.

The parallelised code snippets are shown below.

```
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
```

Listing 6: Inclusion of OMP parallel *do* worksharing constructs within *subroutine* game_of_life().

```
213   ! Calculated the number of cells in each state across the entire grid.
214   ! Inner and outer loops have been swapped due to Fortran storing arrays in
215   ! column-major order.
216   !
217   ! OMP parallel do
218   ! - A static scheduler is chosen as the expected work per iteration should be
219   !   approximately constant. Thus load balancing is unlikely to be an issue,
220   !   while minimising overhead is of particular concern - making a static
221   !   scheduler the optimal choice.
222   ! - The nested loops are not collapsed as testing revealed that including a
223   !   'collapse(2)' clause leads to a ~10% slower execution time.
224   ! - A reduction clause is included for the num_in_state(:) variable as it can
225   !   be reduced across all iterations. Without this clause, the parallel do is
226   !   actually slower than the non-parallel do, however including it leads to
227   !   performance benefits.
228   !
229   !$omp parallel do &
230   !$omp   schedule (static) &
231   !$omp   shared (opt, current_grid) &
232   !$omp   private (i, j, state) &
233   !$omp   reduction(+:num_in_state)
234   do j = 1, opt%m
235     do i = 1, opt%n
236       state = current_grid(i,j)
237       num_in_state(state) = num_in_state(state) + 1
238     end do
239   end do
```

Listing 7: Inclusion of OMP parallel *do* worksharing constructs within *subroutine* game_of_life_stats().

Static schedulers were chosen for the `parallel do` loops, as the amount of work per iteration (of either loop) is expected to be approximately constant. Thus, any benefit from introducing a dynamic or guided scheduler would be minimal and outweighed by the cost of introducing scheduler overhead. A reduction clause is attached to the variable `num_in_state(:)`, since it is effectively an indexed counter which can aggregate the results of each thread together without concern.

It was determined in some small scale tests that including the clause `collapse(2)`, which would collapse the two perfectly-nested loops into one, generally did not produce a significant performance benefit and in some cases worsened performance. However this was only tested rudimentarily and may be more signficant for increasingly large grid sizes.

2.4   Bash Scripts

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 $n_{\text{omp}} = 4$ 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.

```
1   original="output/GOL-01_gol_cpu_serial_fort.nomp-4.ngrid-10x10.nsteps-10.\
2   ic_type-0.vis_type-0.rule_type-0.nghbr_type-0.bndry_type-0/log.txt"
3
4   for log in output/*ngrid-10x10*/log.txt; do
5     diff ${original} ${log}
6   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.

```
1   original="output/GOL-01_gol_cpu_serial_fort.nomp-4.ngrid-10x10.nsteps-10.\
2   ic_type-0.vis_type-0.rule_type-0.nghbr_type-0.bndry_type-0/stats.txt"
3
4   for stats in output/*ngrid-10x10*/stats.txt; do
5     diff ${original} ${stats}
6   done
```

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

3.2   Scaling Behaviour

3.3   Thread Independence Verification

3.4   Thread Scaling Behaviour

A Appendix

A.1 src/common_fort.f90

```fortran
!> 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
```

```fortran
53          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           ! arg length changed from 32 to 2000 to accommodate large statsfilename.
157           character(len=2000) :: arg
158           character(len=2000) :: statsfilename
159           integer :: count
160           integer*8 :: nbytes
161           real*4 :: memfootprint
162           ! get the commands passed and the number of args passed
163           call get_command(cmd)
164           count = command_argument_count()
165           if (count .lt. 2) then
166               write(*,*) "Usage: <grid height> <grid width> "
167               write(*,*) "[<nsteps> <IC type> <Visualisation type> <Rule type> <
      Neighbour type> "
```

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

```fortran
226      end subroutine
227
228      ! get some basic timing info
229      real*8 function init_time()
230          integer, dimension(8) :: value
231          call date_and_time(VALUES=value)
232          init_time = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
233          return
234      end function
235      ! get the elapsed time relative to start
236      subroutine get_elapsed_time(start)
237          real*8, intent(in) :: start
238          real*8 :: finish, delta
239          integer, dimension(8) :: value
240          call date_and_time(VALUES=value)
241          finish = value(5)*3600.0+value(6)*60.0+value(7)+value(8)/1000.0
242          delta = finish - start
243          write(*,*) "Elapsed time is ", delta, "s"
244      end subroutine
245
246 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
```

```fortran
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
```

```fortran
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     ! num_in_state and frac modified from dimension(NUMSTATES) to
127     ! dimension(0:NUMSTATES-1) to synchronise with the 0-based indexing of
128     ! states mandated in common_fort.f90.
129     integer, dimension(0:NUMSTATES-1) :: num_in_state
130     real*4, dimension(0:NUMSTATES-1) :: frac
131     character(len=30) :: fmt
132
133     fmt = "(A15,I1,A3,F10.4,A4)"
134     ntot = opt%n * opt%m
135     num_in_state = 0
136     do i = 1, opt%n
137         do j = 1, opt%m
138             state = current_grid(i,j)
139             num_in_state(state) = num_in_state(state) + 1
140         end do
141     end do
142     frac = num_in_state/real(ntot)
143     if (step .eq. 0) then
144         open(10, file=opt%statsfile, access="sequential")
145     else
146 #if defined(_CRAYFTN) || defined(_INTELFTN)
147         open(10, file=opt%statsfile, position="append")
148 #else
149         open(10, file=opt%statsfile, access="append")
150 #endif
151     end if
152     write(10,*) "step ", step
153     ! Loop bounds modified to synchronise with 0-based indexing of frac(:).
154     do i = 0, NUMSTATES-1
155         write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
156     end do
157     write(10,*) ""
158     close(10)
159 end subroutine
```

A.3   src/02_gol_cpu_serial_fort.f90

```fortran
!> 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.
```

```fortran
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
```

19

```fortran
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
```

```fortran
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   ! num_in_state and frac modified from dimension(NUMSTATES) to
188   ! dimension(0:NUMSTATES-1) to synchronise with the 0-based indexing of
189   ! states mandated in common_fort.f90.
190   integer, dimension(0:NUMSTATES-1) :: num_in_state
191   real*8, dimension(0:NUMSTATES-1) :: frac
192   character(len=30) :: fmt
193
194   fmt = "(A15,I1,A3,F10.4,A4)"
195   ntot = opt%n * opt%m
196
197   num_in_state(:) = 0
198
199   ! Calculated the number of cells in each state across the entire grid.
200   ! Inner and outer loops have been swapped due to Fortran storing arrays in
201   ! column-major order.
202   do j = 1, opt%m
203     do i = 1, opt%n
204       state = current_grid(i,j)
205       num_in_state(state) = num_in_state(state) + 1
206     end do
207   end do
208
209   ! Converted the state occupation from absolute terms to fractional terms.
210   frac(:) = num_in_state(:)/real(ntot)
211
212   if (step .eq. 0) then
213     open(10, file=opt%statsfile, access="sequential")
214   else
215 #if defined(_CRAYFTN) || defined(_INTELFTN)
216     open(10, file=opt%statsfile, position="append")
217 #else
218     open(10, file=opt%statsfile, access="append")
219 #endif
220   end if
221
222   write(10,*) "step ", step
223
224   ! Loop bounds modified to synchronise with 0-based indexing of frac(:).
225   do i = 0, NUMSTATES-1
226     write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
227   end do
228
229   write(10,*) ""
230   close(10)
```

```
231  end subroutine game_of_life_stats
```

A.4   src/02_gol_cpu_openmp_loop_fort.f90

```fortran
!> 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.
```

23

```fortran
 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
```

24

```fortran
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    !$omp end parallel do
189 end subroutine game_of_life
190
191 !> GOL stats
192 subroutine game_of_life_stats(opt, step, current_grid)
193   use gol_common
194   implicit none
195
196   type(Options), intent(in) :: opt
197   integer, intent(in) :: step
198   integer, dimension(:,:), intent(in) :: current_grid
199   integer :: i, j, state
200   integer*8 :: ntot
201   ! num_in_state and frac modified from dimension(NUMSTATES) to
202   ! dimension(0:NUMSTATES-1) to synchronise with the 0-based indexing of
203   ! states mandated in common_fort.f90.
204   integer, dimension(0:NUMSTATES-1) :: num_in_state
205   real*8, dimension(0:NUMSTATES-1) :: frac
206   character(len=30) :: fmt
207
208   fmt = "(A15,I1,A3,F10.4,A4)"
209   ntot = opt%n * opt%m
210
211   num_in_state(:) = 0
212
213   ! Calculated the number of cells in each state across the entire grid.
214   ! Inner and outer loops have been swapped due to Fortran storing arrays in
215   ! column-major order.
216   !
217   ! OMP parallel do
218   ! - A static scheduler is chosen as the expected work per iteration should be
219   !   approximately constant. Thus load balancing is unlikely to be an issue,
220   !   while minimising overhead is of particular concern - making a static
221   !   scheduler the optimal choice.
222   ! - The nested loops are not collapsed as testing revealed that including a
223   !   'collapse(2)' clause leads to a ~10% slower execution time.
224   ! - A reduction clause is included for the num_in_state(:) variable as it can
225   !   be reduced across all iterations. Without this clause, the parallel do is
226   !   actually slower than the non-parallel do, however including it leads to
227   !   performance benefits.
228   !
229   !$omp parallel do &
230   !$omp   schedule (static) &
```

```fortran
231    !$omp    shared (opt, current_grid) &
232    !$omp    private (i, j, state) &
233    !$omp    reduction(+:num_in_state)
234    do j = 1, opt%m
235      do i = 1, opt%n
236        state = current_grid(i,j)
237        num_in_state(state) = num_in_state(state) + 1
238      end do
239    end do
240    !$omp end parallel do
241
242    ! Converted the state occupation from absolute terms to fractional terms.
243    frac(:) = num_in_state(:)/real(ntot)
244
245    if (step .eq. 0) then
246      open(10, file=opt%statsfile, access="sequential")
247    else
248 #if defined(_CRAYFTN) || defined(_INTELFTN)
249      open(10, file=opt%statsfile, position="append")
250 #else
251      open(10, file=opt%statsfile, access="append")
252 #endif
253    end if
254
255    write(10,*) "step ", step
256
257    ! Loop bounds modified to synchronise with 0-based indexing of frac(:).
258    do i = 0, NUMSTATES-1
259      write(10,fmt, advance="no") "Frac in state ", i, " = ", frac(i), " "
260    end do
261
262    write(10,*) ""
263    close(10)
264 end subroutine game_of_life_stats
```

A.5   gol-job-submission.slurm

```bash
#!/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} \
```

```
57  ${rule_type} \
58  ${neighbour_type} \
59  ${boundary_type} \
60  \"${stats_filename}\""
61
62  echo "GOL SLURM job submission"
63  # echo "version_name: ${version_name}"
64  echo "base_name: ${base_name}"
65  # echo "log_filename: ${log_filename}"
66  # echo "stats_filename: ${stats_filename}"
67  # echo "exe: ${exe}"
68  # echo "args: ${args}"
69
70  # check if the GOL simulation has already been performed for these parameters,
71  # and if it hasn't, run the GOL simulation.
72  if [ -d "${output_dir}" ]; then
73      echo "GOL simulation already performed for these parameters"
74  else
75      echo "GOL output directory will be created"
76
77      mkdir -p ${output_dir}
78      touch ${stats_filename}
79      touch ${log_filename}
80
81      echo "GOL simulation will commence"
82
83      srun -n 1 -c ${n_omp} ${exe} ${args} > ${log_filename}
84  fi
```

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

```bash
#!/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

```

30

```bash
57  # GOL job sets
58  echo "GOL SLURM job set submission"
59  echo
60
61  # GOL uniformity / ascii visualisation
62  # Performing GOL simulations on a 10x10 grid, for 10 steps, with ascii
63  # visualisation. Intended to verify that different GOL versions produce the same
64  # behaviour.
65  echo "GOL ascii jobs"
66  echo "versions: ${version_names}"
67  echo "ngrid: 10x10"
68  echo "nsteps: 10"
69  echo "vis_type: 0 (ascii)"
70  for version_name in ${version_names} ; do
71      parameters[version_name]=${version_name}
72      parameters[grid_height]=10
73      parameters[grid_width]=10
74      parameters[num_steps]=10
75      parameters[visualisation_type]=0
76
77      parameter_string=$(kv_string parameters)
78      # echo "--export=${parameter_string}"
79      sbatch --export=${parameter_string} gol-job-submission.slurm
80  done
81  echo
82
83  # GOL scaling
84  # Performing GOL simulations on increasingly large grids, 2^n x 2^n for n = 1,
85  # .., nmax, for 100 steps, with no visualisation. Intended to determine the
86  # scaling behaviour of the different GOL versions.
87  echo "GOL scaling jobs"
88  echo "versions: ${version_names}"
89  echo "grid lengths: ${grid_lengths}"
90  echo "nsteps: 100"
91  echo "vis_type: 3 (none)"
92  for version_name in ${version_names} ; do
93      parameters[version_name]=${version_name}
94      parameters[num_steps]=100
95      parameters[visualisation_type]=3
96
97      for grid_length in ${grid_lengths}; do
98          parameters[grid_height]=${grid_length}
99          parameters[grid_width]=${grid_length}
100
101         parameter_string=$(kv_string parameters)
102         # echo "--export=${parameter_string}"
103         sbatch --export=${parameter_string} gol-job-submission.slurm
104     done
105 done
106 echo
107
108 # GOL thread independence
109 # Performing parallel GOL simulations with varying numbers of omp threads, on a
110 # 10x10 grid, for 10 steps, with ascii visualisation. Intended to verify that
111 # the parallel GOL versions produce the same behaviour independent of the number
112 # of omp threads.
113 echo "GOL thread independence jobs"
114 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
```

A.7   gol-data-extract.sh

```bash
#!/bin/bash -l

# utility function
function parameter_string {
    local -n array=$1

    str=""

    str+="GOL-${array[version_name]}"
    str+=".nomp-${array[n_omp]}"
    str+=".ngrid-${array[grid_height]}x${array[grid_width]}"
    str+=".nsteps-${array[num_steps]}"
    str+=".ic_type-${array[initial_conditions_type]}"
    str+=".vis_type-${array[visualisation_type]}"
    str+=".rule_type-${array[rule_type]}"
    str+=".nghbr_type-${array[neighbour_type]}"
    str+=".bndry_type-${array[boundary_type]}"

    echo ${str}
}

# parameter sets
version_names_serial="01_gol_cpu_serial_fort 02_gol_cpu_serial_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"
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

# GOL scaling batch
parameters[num_steps]=100
parameters[visualisation_type]=3

mkdir -p data/scaling

for version_name in ${version_names} ; do
    parameters[version_name]=${version_name}

    data_file="data/scaling/${version_name}.csv"

    if [ -f ${data_file} ] ; then

        echo "${data_file} already exists"
```

```bash
57
58      else
59
60          for grid_length in ${grid_lengths}; do
61              parameters[grid_height]=${grid_length}
62              parameters[grid_width]=${grid_length}
63
64              str=$(parameter_string parameters)
65
66              echo ${str}
67
68              last_line=$(tail -n 1 output/${str}/log.txt)
69
70              echo ${last_line}
71              time_pattern="([0-9].+)s"
72              if [[ $last_line =~ $time_pattern ]] ; then
73                  time=${BASH_REMATCH[1]}
74                  echo $time
75
76                  printf "${grid_length} , ${time}\n" >> ${data_file}
77              else
78                  echo "failed to find time"
79              fi
80          done
81      fi
82 done
83 echo
84
85 # GOL thread scaling
86 parameters[num_steps]=100
87 parameters[visualisation_type]=3
88
89 mkdir -p data/parallel_thread_scaling
90
91 for version_name in ${version_names_parallel} ; do
92      parameters[version_name]=${version_name}
93
94      for n_omp in ${n_omps}; do
95          parameters[n_omp]=${n_omp}
96
97          data_file="data/parallel_thread_scaling/${version_name}-${n_omp}.csv"
98
99          if [ -f ${data_file} ] ; then
100
101              echo "${data_file} already exists"
102
103          else
104
105              for grid_length in ${grid_lengths}; do
106                  parameters[grid_height]=${grid_length}
107                  parameters[grid_width]=${grid_length}
108
109                  str=$(parameter_string parameters)
110
111                  echo ${str}
112
113                  last_line=$(tail -n 1 output/${str}/log.txt)
114
```

```
115                    echo ${last_line}
116                    time_pattern="([0-9].+)s"
117                    if [[ $last_line =~ $time_pattern ]] ; then
118                        time=${BASH_REMATCH[1]}
119                        echo $time
120
121                        printf "${grid_length} , ${time}\n" >> ${data_file}
122                    else
123                        echo "failed to find time"
124                    fi
125                done
126            fi
127        done
128 done
129 echo
```