OpenMP report

29/04/2024

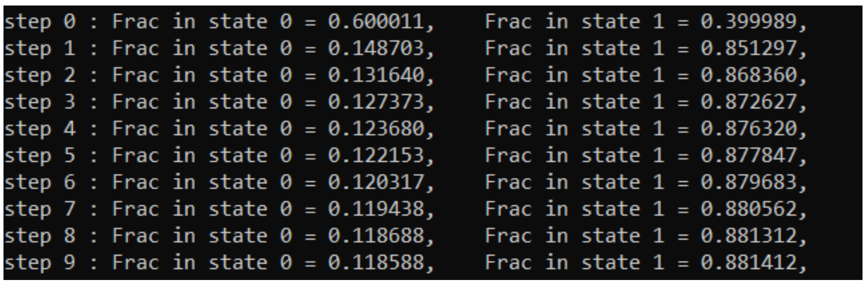
Don Wijesinghe

1.Introduction

In this report we parallelise a 2-D grid based simulation called game of life using OpenMP. In the game, you have a grid of square cells, each of which can be in one of two states: alive or dead. The evolution of the generation of cells is determined by its previous state. For our program the state of the next generation of cells is determined according to a set of rules:

1. Any live cell with fewer than two live neighbours dies, due to underpopulation
2. Any live cell with greater than three live neighbours dies from over population.
3. Any dead cell with exactly three live neighbours becomes a live cell due to reproduction.
4. Any live cell with exactly two live neighbours stays alive for the next generation.

The program uses the initial condition IC = 0 where the grid is spawned with 60% alive and 40% in a specified grid resolution – n x m.

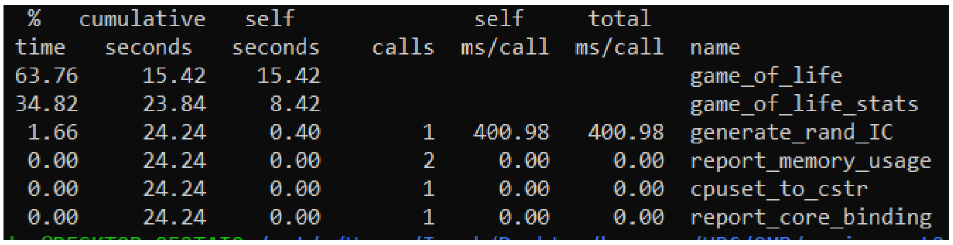


```
step 0 : Frac in state 0 = 0.600011,    Frac in state 1 = 0.399989,
step 1 : Frac in state 0 = 0.148703,    Frac in state 1 = 0.851297,
step 2 : Frac in state 0 = 0.131640,    Frac in state 1 = 0.868360,
step 3 : Frac in state 0 = 0.127373,    Frac in state 1 = 0.872627,
step 4 : Frac in state 0 = 0.123680,    Frac in state 1 = 0.876320,
step 5 : Frac in state 0 = 0.122153,    Frac in state 1 = 0.877847,
step 6 : Frac in state 0 = 0.120317,    Frac in state 1 = 0.879683,
step 7 : Frac in state 0 = 0.119438,    Frac in state 1 = 0.880562,
step 8 : Frac in state 0 = 0.118688,    Frac in state 1 = 0.881312,
step 9 : Frac in state 0 = 0.118588,    Frac in state 1 = 0.881412,
```

Fig1. Data showing the fraction of alive (state 0) and dead (state 1) over 10 steps of evolution in a 10000x10000 grid ran on the serial version.

For this parallelisation we use the C language and the CRAY compiler on a local machine with a ryzen 7 3700x with 8cores and 16threads with 16gb of memory. The code was also run on setonix using a batch script file although all the results and testing shown in this report is from a local machine. The png visualisation does not work in this version of the code and so the *ascii* visualisation was used. The program contains two versions of the program both containing a different parallelisation method – both task loop and regular loop parallelisation. These files are located in the *src* directory. To run the parallelised versions of the program use, the *rungol.sh* script.

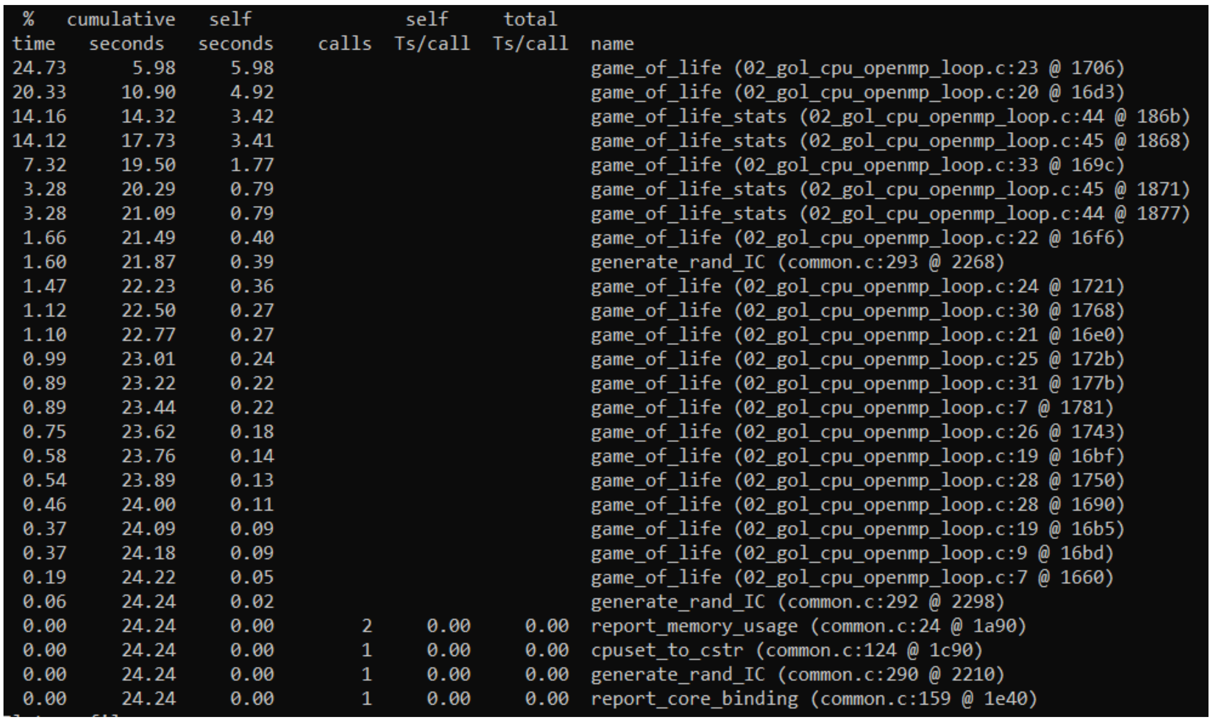
2.Profiling



```
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 63.76     15.42    15.42                             game_of_life
 34.82     23.84     8.42                             game_of_life_stats
  1.66     24.24     0.40        1   400.98   400.98  generate_rand_IC
  0.00     24.24     0.00        2     0.00     0.00  report_memory_usage
  0.00     24.24     0.00        1     0.00     0.00  cpuset_to_cstr
  0.00     24.24     0.00        1     0.00     0.00  report_core_binding
```

Fig2: Profiling of the serial code ran with 10 steps 10000x10000 grid size and no visualisation showing utilisation of time in each function.

Running the serial version of the program, we notice that 64% of the time is spent on the functions *game_of_life* and 35% of the time on the *game_of_life_stats* function. The remaining time is mostly spent on generating the grid with cells using the *generate_rand_IC* function and is only run during the first step. The total elapsed time is 23s where each time step is spending roughly 2.4 seconds each. We also note that *game_of_life* and *game_of_life_stats* both contain iterative double for loops which might be costing the largest amount of performance.



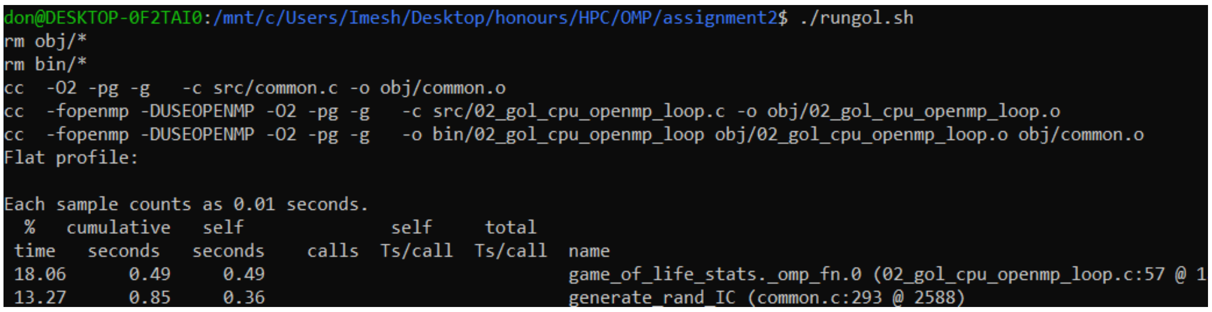
```
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  Ts/call  Ts/call  name    
 24.73      5.98     5.98                             game_of_life (02_gol_cpu_openmp_loop.c:23 @ 1706)
 20.33     10.90     4.92                             game_of_life (02_gol_cpu_openmp_loop.c:20 @ 16d3)
 14.16     14.32     3.42                             game_of_life_stats (02_gol_cpu_openmp_loop.c:44 @ 186b)
 14.12     17.73     3.41                             game_of_life_stats (02_gol_cpu_openmp_loop.c:45 @ 1868)
  7.32     19.50     1.77                             game_of_life (02_gol_cpu_openmp_loop.c:33 @ 169c)
  3.28     20.29     0.79                             game_of_life_stats (02_gol_cpu_openmp_loop.c:45 @ 1871)
  3.28     21.09     0.79                             game_of_life_stats (02_gol_cpu_openmp_loop.c:44 @ 1877)
  1.66     21.49     0.40                             game_of_life (02_gol_cpu_openmp_loop.c:22 @ 16f6)
  1.60     21.87     0.39                             generate_rand_IC (common.c:293 @ 2268)
  1.47     22.23     0.36                             game_of_life (02_gol_cpu_openmp_loop.c:24 @ 1721)
  1.12     22.50     0.27                             game_of_life (02_gol_cpu_openmp_loop.c:30 @ 1768)
  1.10     22.77     0.27                             game_of_life (02_gol_cpu_openmp_loop.c:21 @ 16e0)
  0.99     23.01     0.24                             game_of_life (02_gol_cpu_openmp_loop.c:25 @ 172b)
  0.89     23.22     0.22                             game_of_life (02_gol_cpu_openmp_loop.c:31 @ 177b)
  0.89     23.44     0.22                             game_of_life (02_gol_cpu_openmp_loop.c:7 @ 1781)
  0.75     23.62     0.18                             game_of_life (02_gol_cpu_openmp_loop.c:26 @ 1743)
  0.58     23.76     0.14                             game_of_life (02_gol_cpu_openmp_loop.c:19 @ 16bf)
  0.54     23.89     0.13                             game_of_life (02_gol_cpu_openmp_loop.c:28 @ 1750)
  0.46     24.00     0.11                             game_of_life (02_gol_cpu_openmp_loop.c:28 @ 1690)
  0.37     24.09     0.09                             game_of_life (02_gol_cpu_openmp_loop.c:19 @ 16b5)
  0.37     24.18     0.09                             game_of_life (02_gol_cpu_openmp_loop.c:9 @ 16bd)
  0.19     24.22     0.05                             game_of_life (02_gol_cpu_openmp_loop.c:7 @ 1660)
  0.06     24.24     0.02                             generate_rand_IC (common.c:292 @ 2298)
  0.00     24.24     0.00        2     0.00     0.00  report_memory_usage (common.c:24 @ 1a90)
  0.00     24.24     0.00        1     0.00     0.00  cpuset_to_cstr (common.c:124 @ 1c90)
  0.00     24.24     0.00        1     0.00     0.00  generate_rand_IC (common.c:290 @ 2210)
  0.00     24.24     0.00        1     0.00     0.00  report_core_binding (common.c:159 @ 1e40)
```

Fig 3: Detailed profiling of GOL code within functions for the same initial conditions used in Fig2.

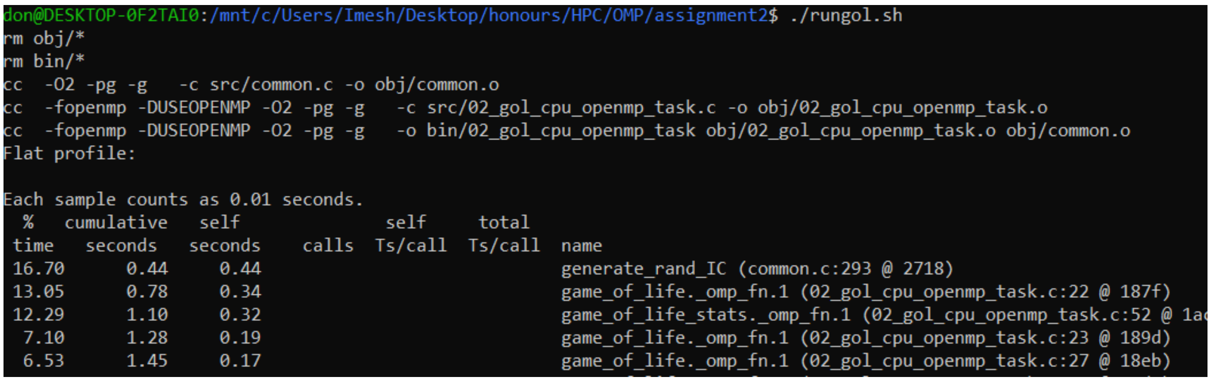
Looking in more detail of the profiling in Fig3, we can see how much time is spent on certain lines of the code within these functions. For the *game_of_life* function we can see that most time is spent on the if conditions where the code accesses the *current_grid* values. Same goes for the *game_of_life_stats* function where it must access the grid within the for loop in line 45. This addresses the need to focus parallelisation within these functions within the loops which will be discussed in section 4.

3.Correctness

The correctness of the parallelised code and the serial version is determined by using visualisation and the stats files produced. Each time the program is run, the output is saved to the *log* directory and the profiling information and stats are sent to the *stats* directory where these output files can be compared. All this is done within the *rungol.sh* script file.



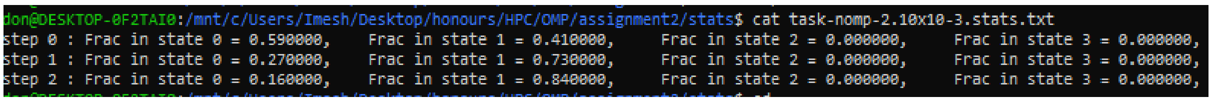
```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2$ ./rungol.sh
rm obj/*
rm bin/*
cc  -O2 -pg -g   -c src/common.c -o obj/common.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -c src/02_gol_cpu_openmp_loop.c -o obj/02_gol_cpu_openmp_loop.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -o bin/02_gol_cpu_openmp_loop obj/02_gol_cpu_openmp_loop.o obj/common.o
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 18.06      0.49     0.49                             game_of_life_stats._omp_fn.0 (02_gol_cpu_openmp_loop.c:57 @ 1
 13.27      0.85     0.36                             generate_rand_IC (common.c:293 @ 2588)
```



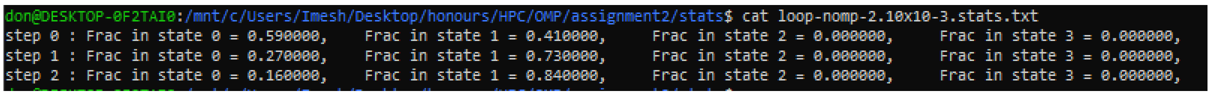
```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2$ ./rungol.sh
rm obj/*
rm bin/*
cc  -O2 -pg -g   -c src/common.c -o obj/common.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -c src/02_gol_cpu_openmp_task.c -o obj/02_gol_cpu_openmp_task.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -o bin/02_gol_cpu_openmp_task obj/02_gol_cpu_openmp_task.o obj/common.o
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 16.70      0.44     0.44                             generate_rand_IC (common.c:293 @ 2718)
 13.05      0.78     0.34                             game_of_life._omp_fn.1 (02_gol_cpu_openmp_task.c:22 @ 187f)
 12.29      1.10     0.32                             game_of_life_stats._omp_fn.1 (02_gol_cpu_openmp_task.c:52 @ 1a
  7.10      1.28     0.19                             game_of_life._omp_fn.1 (02_gol_cpu_openmp_task.c:23 @ 189d)
  6.53      1.45     0.17                             game_of_life._omp_fn.1 (02_gol_cpu_openmp_task.c:27 @ 18eb)
```

The above shows that both the task and loop versions of the program compiles as normal using the *rungol.sh* script.

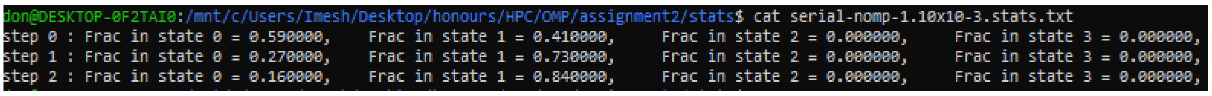
By running both codes on a 10x10 with 3 timesteps and 2 omp threads and not outputting the log files to the directory, we can show that both parallelised versions of the program produce the same visualisation results and stats results as serial.



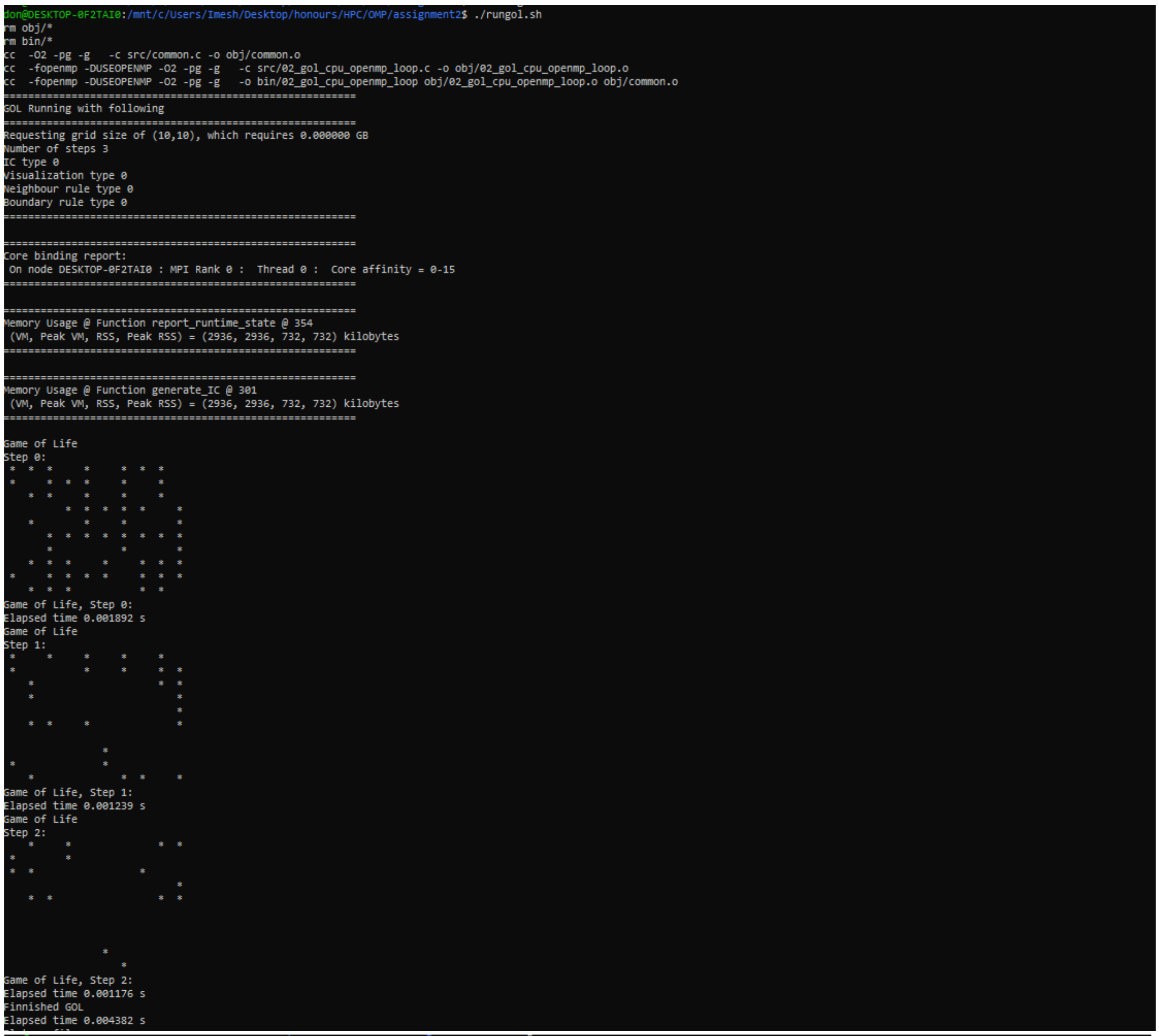
```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2/stats$ cat task-nomp-2.10x10-3.stats.txt
step 0 : Frac in state 0 = 0.590000,    Frac in state 1 = 0.410000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 1 : Frac in state 0 = 0.270000,    Frac in state 1 = 0.730000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 2 : Frac in state 0 = 0.160000,    Frac in state 1 = 0.840000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
```



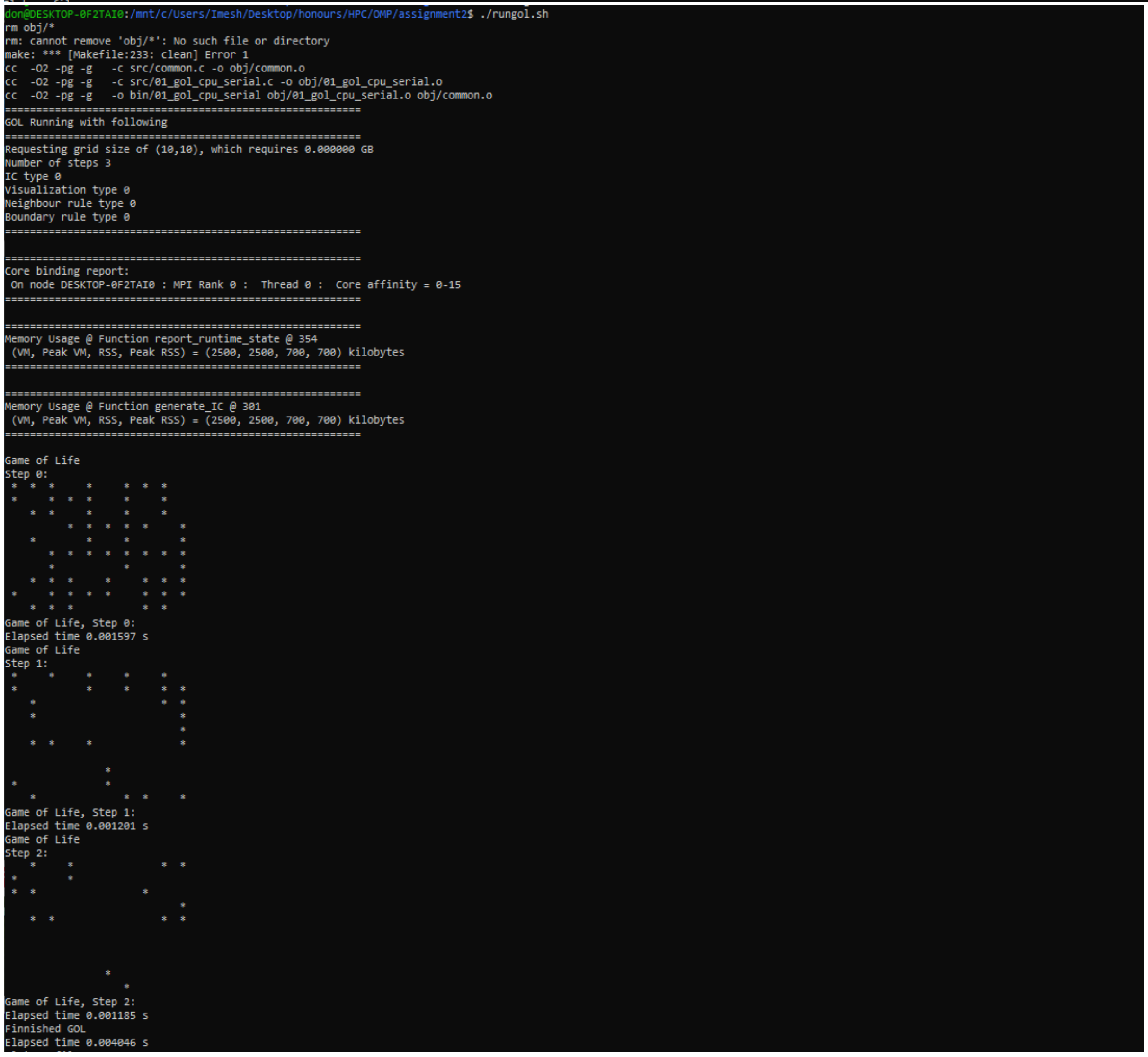
```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2/stats$ cat loop-nomp-2.10x10-3.stats.txt
step 0 : Frac in state 0 = 0.590000,    Frac in state 1 = 0.410000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 1 : Frac in state 0 = 0.270000,    Frac in state 1 = 0.730000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 2 : Frac in state 0 = 0.160000,    Frac in state 1 = 0.840000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
```



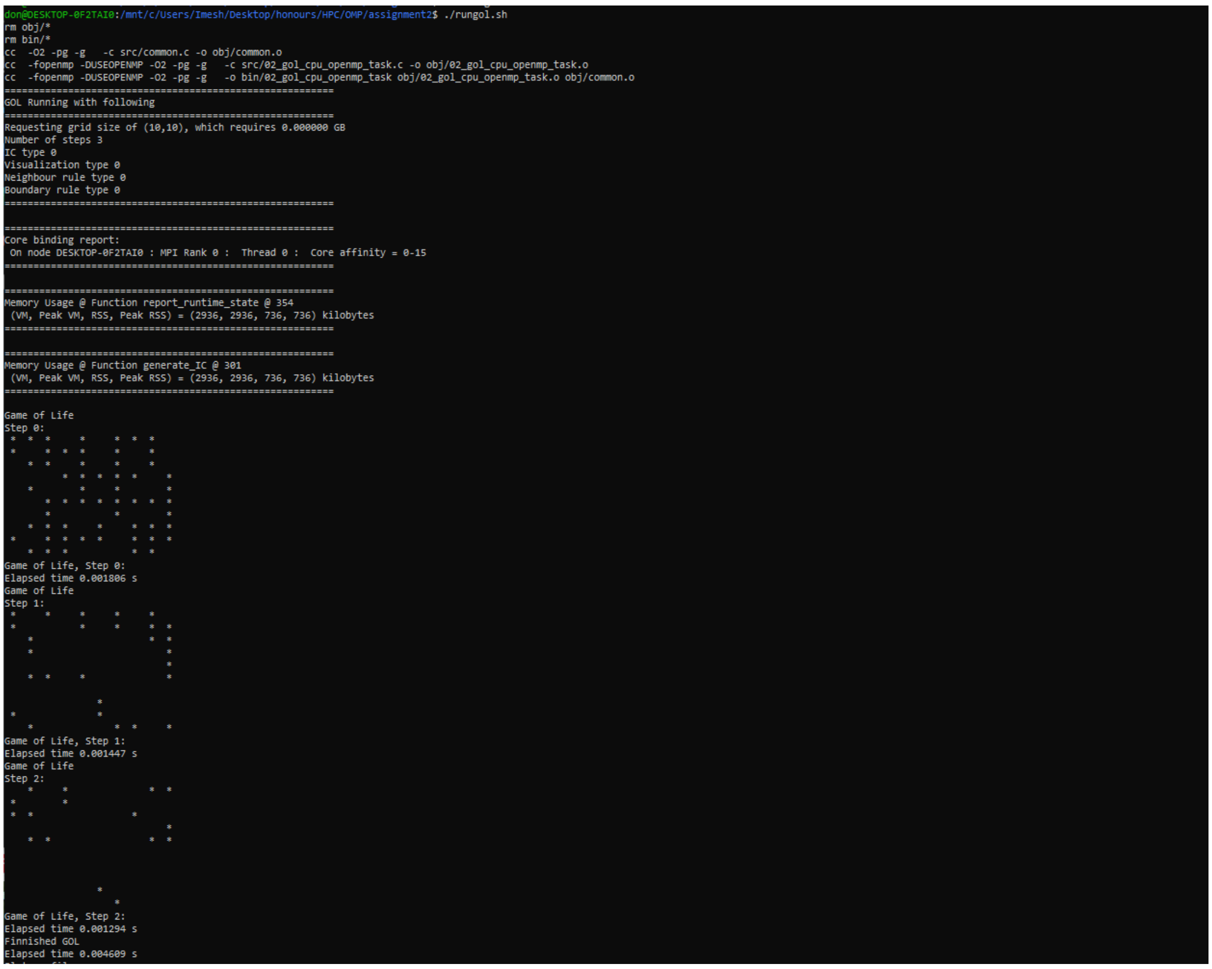
```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2/stats$ cat serial-nomp-1.10x10-3.stats.txt
step 0 : Frac in state 0 = 0.590000,    Frac in state 1 = 0.410000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 1 : Frac in state 0 = 0.270000,    Frac in state 1 = 0.730000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
step 2 : Frac in state 0 = 0.160000,    Frac in state 1 = 0.840000,    Frac in state 2 = 0.000000,    Frac in state 3 = 0.000000,
```



```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2$ ./rungol.sh
rm obj/*
rm bin/*
cc  -O2 -pg -g   -c src/common.c -o obj/common.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -c src/02_gol_cpu_openmp_loop.c -o obj/02_gol_cpu_openmp_loop.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -o bin/02_gol_cpu_openmp_loop obj/02_gol_cpu_openmp_loop.o obj/common.o
=========================================================
GOL Running with following
=========================================================
Requesting grid size of (10,10), which requires 0.000000 GB
Number of steps 3
IC type 0
Visualization type 0
Neighbour rule type 0
Boundary rule type 0
=========================================================

=========================================================
Core binding report:
 On node DESKTOP-0F2TAI0 : MPI Rank 0 :  Thread 0 :  Core affinity = 0-15
=========================================================

=========================================================
Memory Usage @ Function report_runtime_state @ 354
 (VM, Peak VM, RSS, Peak RSS) = (2936, 2936, 732, 732) kilobytes
=========================================================

=========================================================
Memory Usage @ Function generate_IC @ 301
 (VM, Peak VM, RSS, Peak RSS) = (2936, 2936, 732, 732) kilobytes
=========================================================

Game of Life
Step 0:
 *  *  *     *     *  *  *
 *     *  *  *     *     *
    *  *     *     *     *
          *  *  *  *  *     *
    *        *     *        *
       *  *  *  *  *  *  *  *
       *           *        *
    *  *  *     *     *  *  *
 *     *  *  *  *     *  *  *
    *  *  *           *  *
Game of Life, Step 0:
Elapsed time 0.001892 s
Game of Life
Step 1:
 *     *     *     *     *
 *           *     *     *  *
    *                    *  *
    *                       *
                            *
    *  *     *              *

                *
 *              *
    *              *  *     *
Game of Life, Step 1:
Elapsed time 0.001239 s
Game of Life
Step 2:
    *     *              *  *
 *        *
 *  *                 *
                            *
    *  *                 *  *



                *
                   *
Game of Life, Step 2:
Elapsed time 0.001176 s
Finnished GOL
Elapsed time 0.004382 s
```



```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2$ ./rungol.sh
rm obj/*
rm: cannot remove 'obj/*': No such file or directory
make: *** [Makefile:233: clean] Error 1
cc  -O2 -pg -g   -c src/common.c -o obj/common.o
cc  -O2 -pg -g   -c src/01_gol_cpu_serial.c -o obj/01_gol_cpu_serial.o
cc  -O2 -pg -g   -o bin/01_gol_cpu_serial obj/01_gol_cpu_serial.o obj/common.o
=========================================================
GOL Running with following
=========================================================
Requesting grid size of (10,10), which requires 0.000000 GB
Number of steps 3
IC type 0
Visualization type 0
Neighbour rule type 0
Boundary rule type 0
=========================================================

=========================================================
Core binding report:
 On node DESKTOP-0F2TAI0 : MPI Rank 0 :  Thread 0 :  Core affinity = 0-15
=========================================================

=========================================================
Memory Usage @ Function report_runtime_state @ 354
 (VM, Peak VM, RSS, Peak RSS) = (2500, 2500, 700, 700) kilobytes
=========================================================

=========================================================
Memory Usage @ Function generate_IC @ 301
 (VM, Peak VM, RSS, Peak RSS) = (2500, 2500, 700, 700) kilobytes
=========================================================

Game of Life
Step 0:
 *  *  *     *     *  *  *
 *     *  *  *     *     *
    *  *     *     *     *
          *  *  *  *  *     *
    *        *     *        *
       *  *  *  *  *  *  *  *
       *           *        *
    *  *  *     *     *  *  *
 *     *  *  *  *     *  *  *
    *  *  *           *  *
Game of Life, Step 0:
Elapsed time 0.001597 s
Game of Life
Step 1:
 *     *     *     *     *
 *           *     *     *  *
    *                    *  *
    *                       *
                            *
    *  *     *              *

                *
 *              *
    *              *  *     *
Game of Life, Step 1:
Elapsed time 0.001201 s
Game of Life
Step 2:
    *     *              *  *
 *        *
 *  *                 *
                            *
    *  *                 *  *



                *
                   *
Game of Life, Step 2:
Elapsed time 0.001185 s
Finnished GOL
Elapsed time 0.004046 s
```



```
don@DESKTOP-0F2TAI0:/mnt/c/Users/Imesh/Desktop/honours/HPC/OMP/assignment2$ ./rungol.sh
rm obj/*
rm bin/*
cc  -O2 -pg -g   -c src/common.c -o obj/common.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -c src/02_gol_cpu_openmp_task.c -o obj/02_gol_cpu_openmp_task.o
cc  -fopenmp -DUSEOPENMP -O2 -pg -g   -o bin/02_gol_cpu_openmp_task obj/02_gol_cpu_openmp_task.o obj/common.o
==========================================================
GOL Running with following
==========================================================
Requesting grid size of (10,10), which requires 0.000000 GB
Number of steps 3
IC type 0
Visualization type 0
Neighbour rule type 0
Boundary rule type 0
==========================================================

==========================================================
Core binding report:
 On node DESKTOP-0F2TAI0 : MPI Rank 0 :  Thread 0 :  Core affinity = 0-15
==========================================================

==========================================================
Memory Usage @ Function report_runtime_state @ 354
 (VM, Peak VM, RSS, Peak RSS) = (2936, 2936, 736, 736) kilobytes
==========================================================

==========================================================
Memory Usage @ Function generate_IC @ 301
 (VM, Peak VM, RSS, Peak RSS) = (2936, 2936, 736, 736) kilobytes
==========================================================

Game of Life
Step 0:
 * * *   *   * * *
 *   * * *   *   *
   * *   *   *   *
       * * * * *   *
   *     *   *     *
     * * * * * * * *
     *       *     *
   * * *   *   * * *
 *   * * * *   * * *
   * * *       * *
Game of Life, Step 0:
Elapsed time 0.001806 s
Game of Life
Step 1:
 *   *   *   *   *
 *       *   *   * *
   *             * *
   *               *
                   *
   * *   *         *

           *
 *         *
   *         * *   *
Game of Life, Step 1:
Elapsed time 0.001447 s
Game of Life
Step 2:
   *   *         * *
 *     *
 * *           *
                   *
   * *           * *



           *
             *
Game of Life, Step 2:
Elapsed time 0.001294 s
Finnished GOL
Elapsed time 0.004609 s
```

The diff command was also used to compare log files to see if the visualisation was the same for different runs, especially for larger grid sizes where its difficult to confirm the output is the same by eye.

4.Implementation and Code

In this section we look into the OpenMP parallelism implemented to the serial version of the code where we utilise task and loop parallelism and discuss why certain choices are made. We also look into optimisation of the serial code. In this section we will use a 10000x10000 grid with 10steps and for testing and comparisons.

4.1 serial optimisation

As seen in section 2, the serial version of the code is poorly optimised where the for loops in both functions take up a significant amount of time. The entire 2-D grid is contained within a 1D array. Having j as the outerloop and i as the inner loop causes the array to be accessed in a non-contiguous manner. We need to organise our code such that our code can read data as fast as possible. One way of doing this is utilising memory access so that you benefit from cache memory since reading from main memory is very slow. Most of the memory reads should refer to elements that you have recently read. Hence swapping the loops around would avoid cache misses since data accessed in one iteration are likely to be accessed again in the next iteration.

```
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

   ...


   for(int i=0; i < n; i++){
       for(int j=0; j<m; j++){

    ...

       }
    }
}
```

Fig 4: *game_of_life* function with optimised loop ordering to avoid cache misses.

Now, the inner loop iterates over the j index, ensuring that adjacent elements of the array are accessed in adjacent iterations.This change is also applied to the for loop in *game_of_life_stats* function.

After this change we see significant improvements in each function's execution times – 4.9sec and 1.43 seconds for *game_of_life* and *game_of_life_stats* respectively. Which is 3 folds faster and nearly 6 folds faster for the *game_of_life* and *game_of_life_stats* respectively compared to before. Now the total execution time is distributed 72% to *game_of_life* function and 21% to *game_of_life_stats* function with a significant improvement in the total elapsed time of ~6.3seconds compared to ~23seconds before.

Furthermore, changing the double loop to a single loop using the modular method does speed it up significantly however still about 2 seconds slower than having the correct loop order.

```
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

...

int grid_size = n*m;
int i,j;

for(int loop = 0; loop < grid_size; loop++){

        // calculate indices i and j
        i = loop/m;
        j = loop%m;

  ...

 }
```

Fig 5: *game_of_life* function loop utilising a modular method.

Attempting to optimise the code inside the for loop proved to be unnecessary and was kept the same throughout the entire parallelism. In the code snippets shown below we attempted to calculate the neighbouring indices and number of surrounding neighbours using a different approach, which both proved to be slower. Having the loop version for finding the neighbour indices shown in Fig 6 almost doubles the time spent inside the *game_of_life_loop* function. finding the indices in the original serial version is much faster where it took 4.8 secs in total inside the function and 10.2 seconds when changed to the loop version. Since most of the time is still being spent on the if statements, we attempted to use a double for loop shown in Fig 7 to calculate the number of neighbours, however it made it slower than the serial version after our loop ordering change.

```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

...

/*create an offset array*/
int n_i_offset[8] = {-1,-1,-1,0,1,1,1,0};
int n_j_offset[8] = {-1,0,1,1,1,0,-1,-1};
for(int i = 0; i < n; i++){
    for(int j = 0; j < m; j++){
        // count the number of neighbours, clockwise around the current cell.
        neighbours = 0;
        for (int k = 0; k < 8; k++){

                n_i[k] = i + n_i_offset[k];
                n_j[k] = j + n_j_offset[k];
        }
  ...

 }
```

Fig 6: alternative approach to calculating the neighbour indices.

```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

   ...

   for(int k = -1; k <= 1; k++) {
      for (int l = -1; l <= 1; l++) {
           if (!(k == 0 && l == 0) && (i + k >= 0 && i + k < n) && (j + l >= 0 && j + l < m))
           {
                neighbours += current_grid[(i+k)*m +(j+l)];
           }
      }
   }

   ...

}
```

Fig 7: alternative approach to calculating the number of neighbours around the current cell.

4.2.Loop parallelisation

Since we can no longer optimise the serial version further, we now implement omp parallelism. The best choice of parallelism is to parallelise the for loops in both *game_of_life* and *game_of_life_stats* as these takes the majority of time in the program. Looking at the code in *02_cpu_golopenmp_loop.c* in Fig 8, The code begins with a parallel region defined by #pragma omp parallel for. This directive instructs the compiler to parallelize the subsequent loop across multiple threads. Each thread executes the loop independently, with iterations distributed among them. In this implementation each thread within the parallel region is assigned a subset of the outside loop iterations to process concurrently, but not inside. Theres also the case where since the outermost loop is parallelised, the number of iterations of ‘*i*’ could be lower than the number of threads. For cases like this we could either parallelise the inner loop by using the *collapse(2)* clause as seen in Fig 10 or we can use the modular method, both at the cost of being a little slower (~0.15s).

```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){
    int neighbours;
    int n_i[8], n_j[8];
    //loop is broken up and devided amongst threads to complete work using a specified
    //scheduling method during runtime.
    #pragma omp parallel for \
    default(none) shared(next_grid,current_grid,n,m,opt) private(neighbours, n_i, n_j) \
    schedule(runtime)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            neighbours = 0;
            //collect neighbouring cell's indices.
            n_i[0] = i - 1; n_j[0] = j - 1;
            n_i[1] = i - 1; n_j[1] = j;
            n_i[2] = i - 1; n_j[2] = j + 1;
            n_i[3] = i;     n_j[3] = j + 1;
            n_i[4] = i + 1; n_j[4] = j + 1;
            n_i[5] = i + 1; n_j[5] = j;
            n_i[6] = i + 1; n_j[6] = j - 1;
            n_i[7] = i;     n_j[7] = j - 1;



            // count the number of neighbours, clockwise around the current cell.
            if(n_i[0] >= 0 && n_j[0] >= 0 && current_grid[n_i[0] * m + n_j[0]] == ALIVE)
            neighbours++;
            if(n_i[1] >= 0 && current_grid[n_i[1] * m + n_j[1]] == ALIVE) neighbours++;
            if(n_i[2] >= 0 && n_j[2] < m && current_grid[n_i[2] * m + n_j[2]] == ALIVE)
            neighbours++;
            if(n_j[3] < m && current_grid[n_i[3] * m + n_j[3]] == ALIVE) neighbours++;
            if(n_i[4] < n && n_j[4] < m && current_grid[n_i[4] * m + n_j[4]] == ALIVE)
            neighbours++;
            if(n_i[5] < n && current_grid[n_i[5] * m + n_j[5]] == ALIVE) neighbours++;
            if(n_i[6] < n && n_j[6] >= 0 && current_grid[n_i[6] * m + n_j[6]] == ALIVE)
            neighbours++;
            if(n_j[7] >= 0 && current_grid[n_i[7] * m + n_j[7]] == ALIVE) neighbours++;

            //depending on the number of neighbours, decide if the current cell should live or
            //die
            if(current_grid[i*m + j] == ALIVE && (neighbours == 2 || neighbours == 3)){
                next_grid[i*m + j] = ALIVE;
            } else if(current_grid[i*m + j] == DEAD && neighbours == 3){
                next_grid[i*m + j] = ALIVE;
            }else{
                next_grid[i*m + j] = DEAD;
            }
        }
    }
}
```

Fig 8: loop parallelisation using omp for *game_of_life* function.

```c
void game_of_life_stats(struct Options *opt, int step, int *current_grid){
    unsigned long long num_in_state[NUMSTATES];
    int m = opt->m, n = opt->n;
    for(int i = 0; i < NUMSTATES; i++) num_in_state[i] = 0;

    //each thread counts cells in each state for assigned part of the loop and does a reductior
    //into the master thread by doing a sum operation.
    #pragma omp parallel for reduction(+: num_in_state) schedule(runtime)
    for(int loop = 0; loop < n*m; loop++){
            num_in_state[current_grid[loop]]++;

    }

    double frac, ntot = opt->m*opt->n;
    FILE *fptr;
    if (step == 0) {
        fptr = fopen(opt->statsfile, "w");
    }
    else {
        fptr = fopen(opt->statsfile, "a");
    }
    fprintf(fptr, "step %d : ", step);
    for(int i = 0; i < NUMSTATES; i++) {
        frac = (double)num_in_state[i]/ntot;
        fprintf(fptr, "Frac in state %d = %f,\t", i, frac);
    }
    fprintf(fptr, " \n");
    fclose(fptr);
}
```

Fig 9: loop parallelisation using omp for *game_of_life_stats* function.

```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

    ...

    #pragma omp parallel for \
    default(none) shared(next_grid,current_grid,n,m,opt) private(neighbours, n_i, n_j) \
    collapse(2)

    for(int i=0; i < n; i++){
        for(int j=0; j<m; j++){

     ...

        }
     }
}
```

Fig 10: loop parallelisation using *collapse()* clause, so that both loops are parallelised.

```
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){

   ...

   #pragma omp parallel for \
   default(none) shared(next_grid,current_grid,n,m,opt) private(neighbours, n_i, n_j)

   for(int loop = 0; loop < n*m; loop++){
       i=loop/n;
       j=loop%n;

    ...

}
```

Fig 11. loop parallelisation using the modular method, which also achieves the same result as using *collapse()*.

As our main implementation of loop parallelisation, we have kept the code as is in Fig 8 as this achieves the fastest execution time, where it achieves a total elapsed time of ~0.91s.

Furthermore, the loop variables i and j are private to each thread by default, ensuring that each thread maintains its own copy of these variables during loop execution. The shared clause in the #pragma omp parallel for directive specifies that variables *next_grid, current_grid, n, m*, and opt are shared among all threads. This means that these variables are accessible and modifiable by all threads in the parallel region, although *current_grid* is not changed inside this function, we still need to access it's data, whereas setting it private would set all values to 0 once entering parallel region. The private clause specifies that the variables neighbours*, n_i*, and *n_j* are private to each thread. Each thread has its own copy of these variables, ensuring that they are not shared among threads and avoiding potential data races. Inside the nested loops, each iteration performs an independent computation to determine the next state of the cells in the grid. The computations within each iteration can be executed in parallel by multiple threads, allowing for efficient parallel processing of the entire grid.

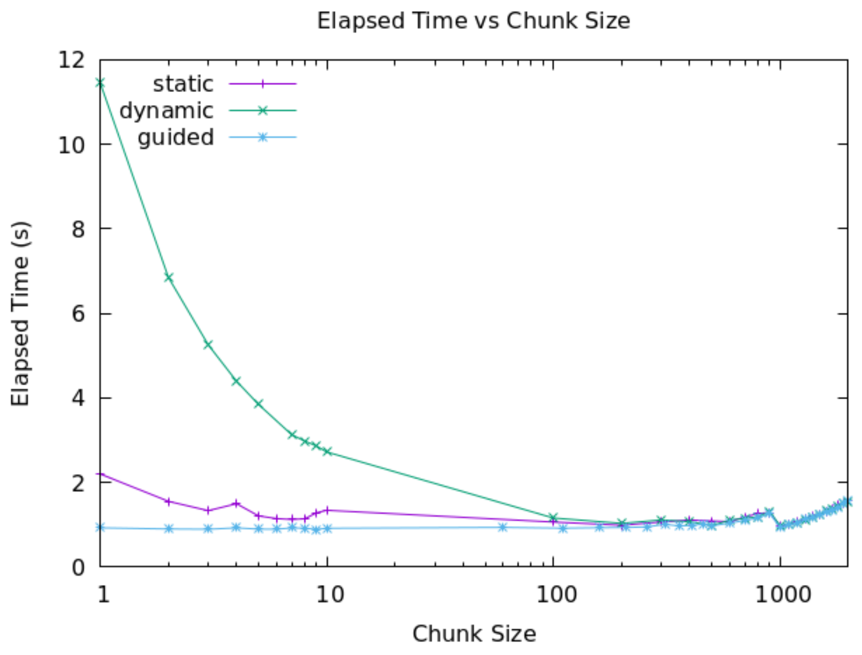
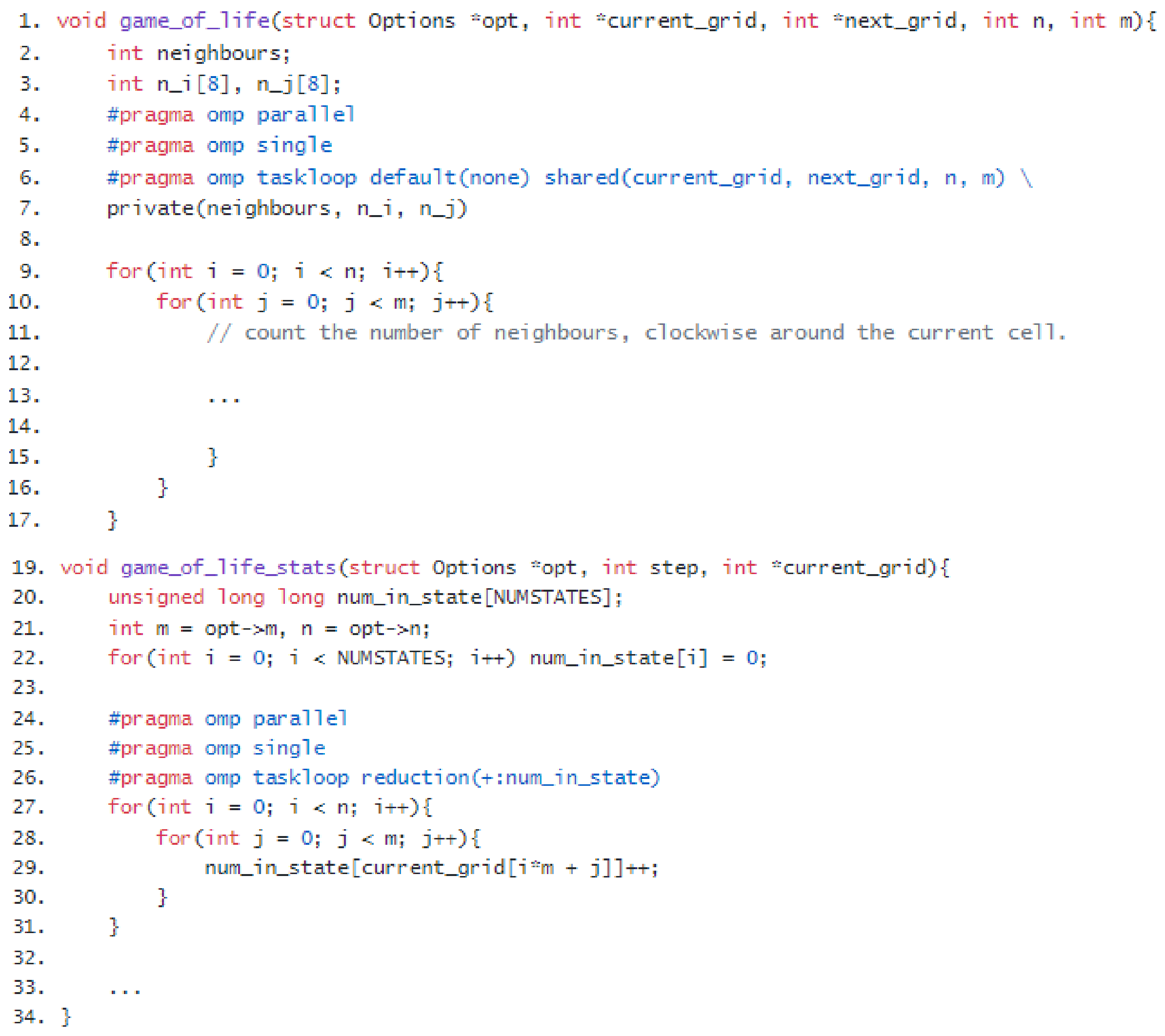


Fig 12: illustrates the relationship between chunk size and total elapsed time for both static and dynamic scheduling, as well as the impact of minimum chunk size on elapsed time for guided scheduling.

The schedule(runtime) clause specifies that the scheduling of loop iterations should be determined at runtime by the OpenMP runtime system where you can specify either dynamic, static or guided scheduling. In Fig 12 we see that the best scheduling to be used is guided scheduling as it achieves the lowest runtime for smaller minimum chunk sizes. By specifying a minimum chunk size parameter in guided scheduling, it dynamically adjusts the chunk size based on the number of remaining iterations and the number of threads available and this chunk size decreases exponentially until it reaches the minimum chunk size and each thread processes this minimum chunk size until the loop finishes. We can see that guided achieves a minimum overhead compared to the other methods. Dynamic scheduling has high communication overhead as the chunk size decreases causing threads to spend more time communicating over doing work. For higher chunk sizes we see that all lines start to converge and elapsed time starts to increase after the chunk size n/(number of threads).

4.3.Task parallelisation

Similarly for task parallelisation we parallelise the same for loops using the taskloop directive as shown below in Fig 13.

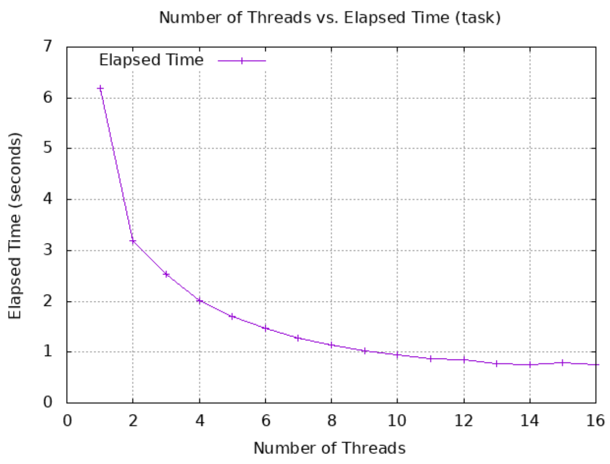


```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){
    int neighbours;
    int n_i[8], n_j[8];
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop default(none) shared(current_grid, next_grid, n, m) \
    private(neighbours, n_i, n_j)

    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            // count the number of neighbours, clockwise around the current cell.

            ...

            }
        }
    }

void game_of_life_stats(struct Options *opt, int step, int *current_grid){
    unsigned long long num_in_state[NUMSTATES];
    int m = opt->m, n = opt->n;
    for(int i = 0; i < NUMSTATES; i++) num_in_state[i] = 0;

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop reduction(+:num_in_state)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            num_in_state[current_grid[i*m + j]]++;
        }
    }

    ...
}
```

Fig 13: showcases the implementation of task loop parallelism, applied to both *game_of_life* and *game_of_life_stats* for loops.

The *taskloop* directive is used to parallelise loops that generate multiple independent tasks. Its similar to parallel for, but instead of generating threads to execute iterations of the loop in parallel, it generates tasks. Inside the loop, tasks are generated for each iteration. These tasks represent the work to be done in each iteration of the loop. OpenMP runtime manages these tasks and schedules them across available threads. It turns out that task loop parallelism has similar performance in comparison to loop parallelism but a little faster, ~3%.

5.Scaling



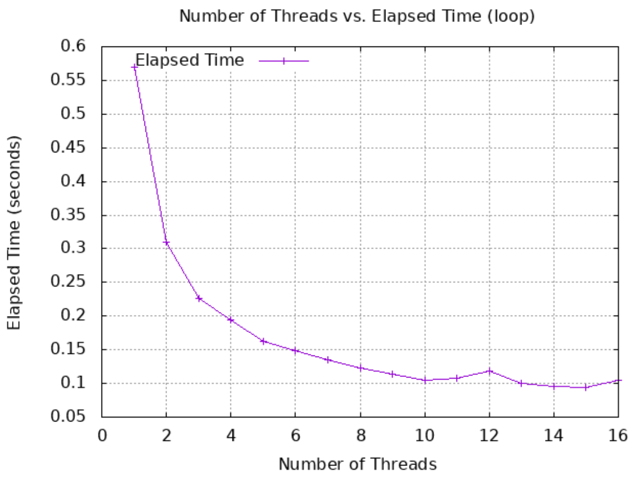
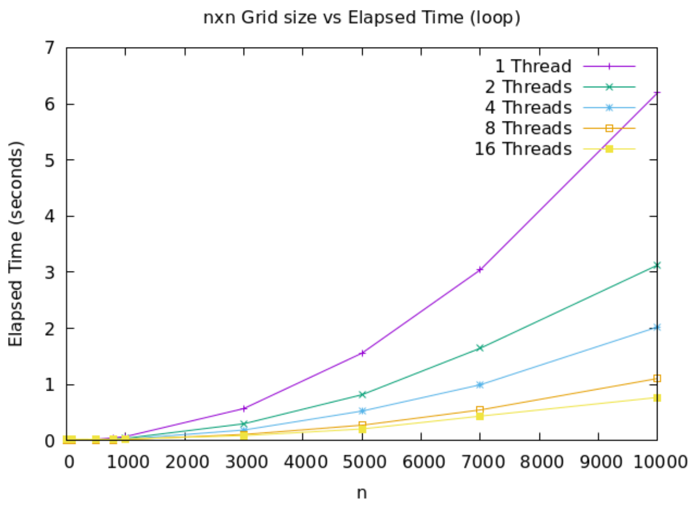


Fig 14: illustrates the relationship between the number of threads and elapsed time for a grid size of 10000x10000 using 10 steps for task parallelism, and 1000x1000 using 5steps for loop parallelism using guided scheduling using no chunk size input. The graph demonstrates a noticeable decrease in elapsed time as the number of threads increases.



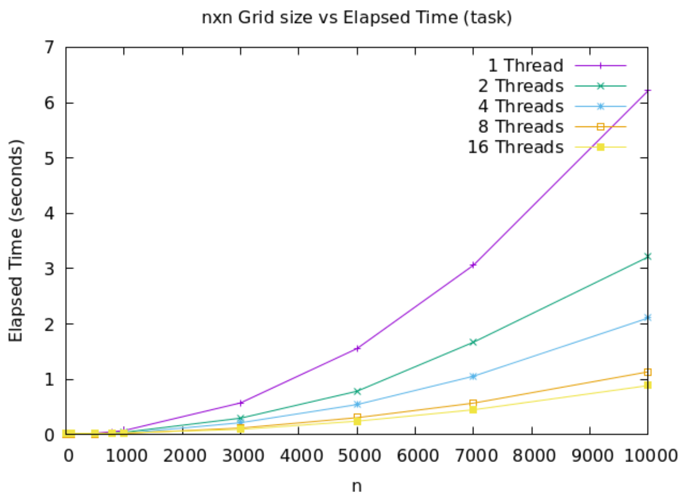


Fig 15. Illustrates the relationship between increase in grid size and elapsed time for different number of threads for 10000x10000 grid size and 10 steps with guided scheduling using no minimum chunk size input.

As the size of the grid increases, the computational complexity of simulating each generation also increases. This can lead to longer execution times, especially for larger grids, as there are more cells to update. the overall time complexity for simulating one generation can be expressed as O(n^2), where n is the size of one dimension of the grid. This is because there are n x n cells in the grid, and each cell must be updated once per generation. This O(n^2) behaviour is seen in Fig 15.

Fig:14 shows the elapsed time vs number of threads, for both task and loop parallelism with different grid sizes and number of steps. We notice that the relationship between the time and number of threads are the same. The total elapsed time decreases with increasing counts of threads and has the for a particular grid size the scaling behaviour is O(N/t) where t is the number of threads and N is the grid size N = nxn. It was found that the other scheduling methods – static and dynamic also had this same behaviour at their default chunk size value (n/t). The same scaling behaviour applies to running with no scheduling at all. When considering arbitrary chunk sizes as inputs for scheduling we would still get the same scaling behaviour(shape) however we know that communication overhead becomes significant at lower chunk sizes for dynamic scheduling as seen in Fig 12. Hence the scaling graph would be shifted upwards due to having higher execution times compared to static and guided scheduling.