HPC

Assignment 2

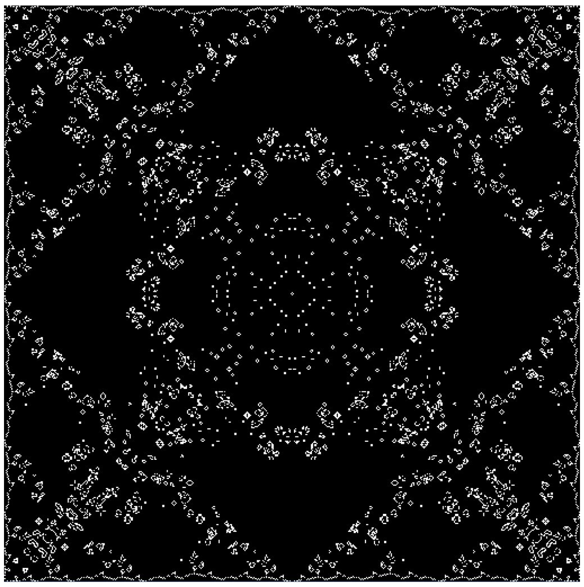
Curtin University

Alec Dorrington

Student ID 19133047

Table of Contents

[1] Game of Life



Source code

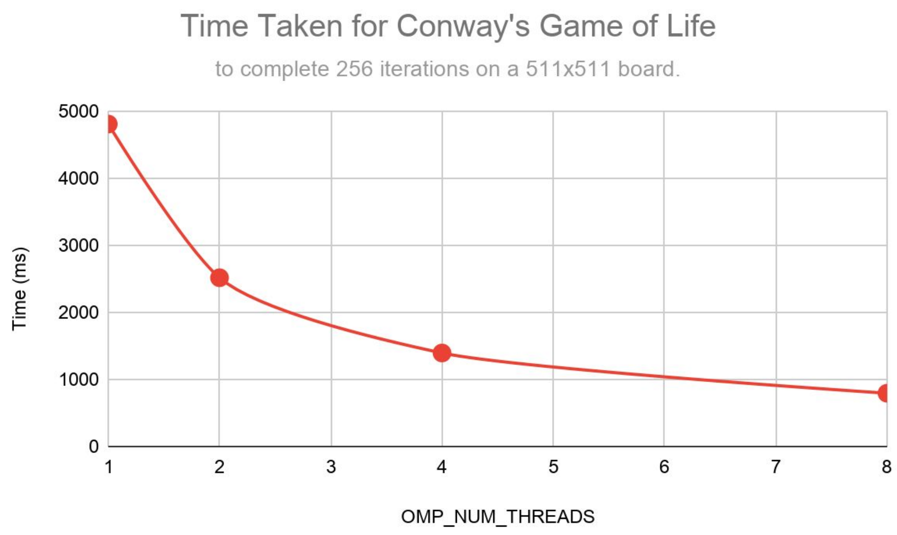
```c
// Conway's Game of Life simulator, using OpenMP.
// Written by Alec Dorrington.
// https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>

// Horizontal and vertical grid size.
#define SIZE 511
// Number of game iterations to simulate.
#define ITERATIONS 256
// Pixel width of each grid square.
#define SCALE 2
// Name of file(s) to write output image(s).
#define FILENAME "life%3.3d.pgm"
// Chunk size (number of rows) used for dynamic decomposition.
#define CHUNKSIZE 1

// Modulo function which works for negative numbers.
#define MOD(a, b) (((a) + (b)) % (b))

// Function type used to generate initial board configurations.
// Given a grid size and (x, y) coordinate,
// return whether this position is filled (0=no, 1=yes).
```

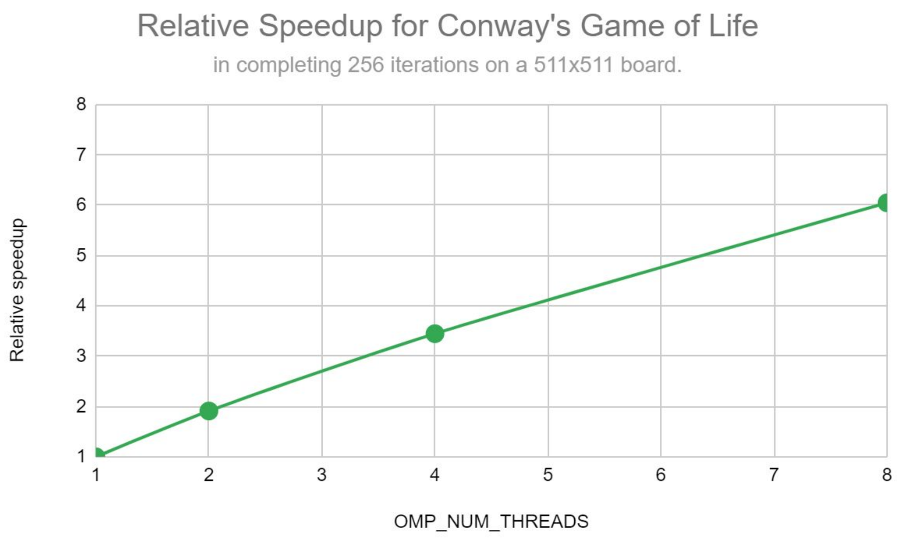
```c
typedef int (*generator_t)(int, int, int);

// Write a Game of Life grid to an image file.
void writePgm(int **grid, int size, int scale, char *fileName) {

  int x, y;
  FILE *file = fopen(fileName, "w");

  // Write the image header specifying dimensions.
  fprintf(file, "P2\n%4d %4d\n1\n", size * scale, size * scale);

  // Write each pixel in the output image.
  for(x = 0; x < size * scale; x++) {
    for(y = 0; y < size * scale; y++) {
      fprintf(file, "%d\n", grid[x / scale][y / scale]);
    }
  }
  fclose(file);
}

// Create a size x size grid where f specifies the original configuration.
// Filled squares are represented by a 1, while empty squares are 0.
int** createGrid(int size, generator_t f) {

  int x, y;
  // Allocate the outer array.
  int **grid = malloc(size * sizeof(int*));

  // Allocate each inner array.
  for(x = 0; x < size; x++) {
    grid[x] = malloc(size * sizeof(int));

    // Use f to decide whether each square is empty or filled.
    for(y = 0; y < size; y++) {
      grid[x][y] = f(size, x, y);
    }
  }
  return grid;
}

// Free the memory allocated for a grid.
void freeGrid(int **grid, int size) {

  int x;
  for(x = 0; x < size; x++) {
    free(grid[x]);
  }
  free(grid);
}
```

```c
// Determine how many of the 8 adjacent squares are filled.
// Uses cyclic boundary conditions so that the grid is wrapped like a torus.
int adjacent(int **grid, int size, int x, int y) {

  int xx, yy, adj = 0;

  // For each adjacent square.
  for(xx = -1; xx <= 1; xx++) {
    for(yy = -1; yy <= 1; yy++) {

      // Count each filled square which isn't this one.
      if(xx != 0 || yy != 0) {
        adj += grid[MOD(x + xx, size)][MOD(y + yy, size)];
      }
    }
  }
  return adj;
}

// Perform a single iteration of the Game of Life.
// Takes the state of grid1 and puts the next state in grid2.
void life(int **grid1, int **grid2, int size) {

  int x, y, adj;

  // For each grid square (in parallel).
  #pragma omp parallel for \
    shared(grid1, grid2, size) private(x, y, adj) \
    schedule(dynamic, CHUNKSIZE)
  for(x = 0; x < size; x++) {
    for(y = 0; y < size; y++) {

      // Determine the number of filled adjacent squares.
      adj = adjacent(grid1, size, x, y);

      // Progress each square according to the Game of Life rules.
      if(adj < 2 || adj > 3) grid2[x][y] = 0;
      if(adj == 2) grid2[x][y] = grid1[x][y];
      if(adj == 3) grid2[x][y] = 1;
    }
  }
}

// Function to generate an empty Game of Life grid.
int EMPTY(int size, int x, int y) {
  return 0;
}

// Function to generate a Game of Life grid with a simple cross.
int CROSS(int size, int x, int y) {
  return x == size / 2 || y == size / 2;
}

// Get the number of milliseconds since the Unix epoch.
long timeMs() {
  struct timeval time;
  gettimeofday(&time, NULL);
  return time.tv_sec * 1000 + time.tv_usec / 1000;
}

int main(int argc, int **argv) {

  int i; long t;
  char fileName[20];
  int **grid1 = createGrid(SIZE, CROSS),
    **grid2 = createGrid(SIZE, EMPTY), **swap;

  // Write the initial grid configuration to an image file.
  sprintf(fileName, FILENAME, 0);
  writePgm(grid1, SIZE, SCALE, fileName);

  t = timeMs();

  // Continue incrementing the grid state for some number of iterations.
  for(i = 0; i < ITERATIONS; i++) {
    life(grid1, grid2, SIZE);
    // Swap grid1 and grid2 so that grid1 is always the most recent.
    swap = grid1; grid1 = grid2; grid2 = swap;
  }

  printf("Completed %d Game of Life iterations on a %dx%d grid in %dms.\n",
    ITERATIONS, SIZE, SIZE, timeMs() - t);

  // Write the final grid configuration to an image file.
  sprintf(fileName, FILENAME, ITERATIONS);
  writePgm(grid1, SIZE, SCALE, fileName);

  freeGrid(grid1, SIZE); freeGrid(grid2, SIZE);
  return 0;
}
```

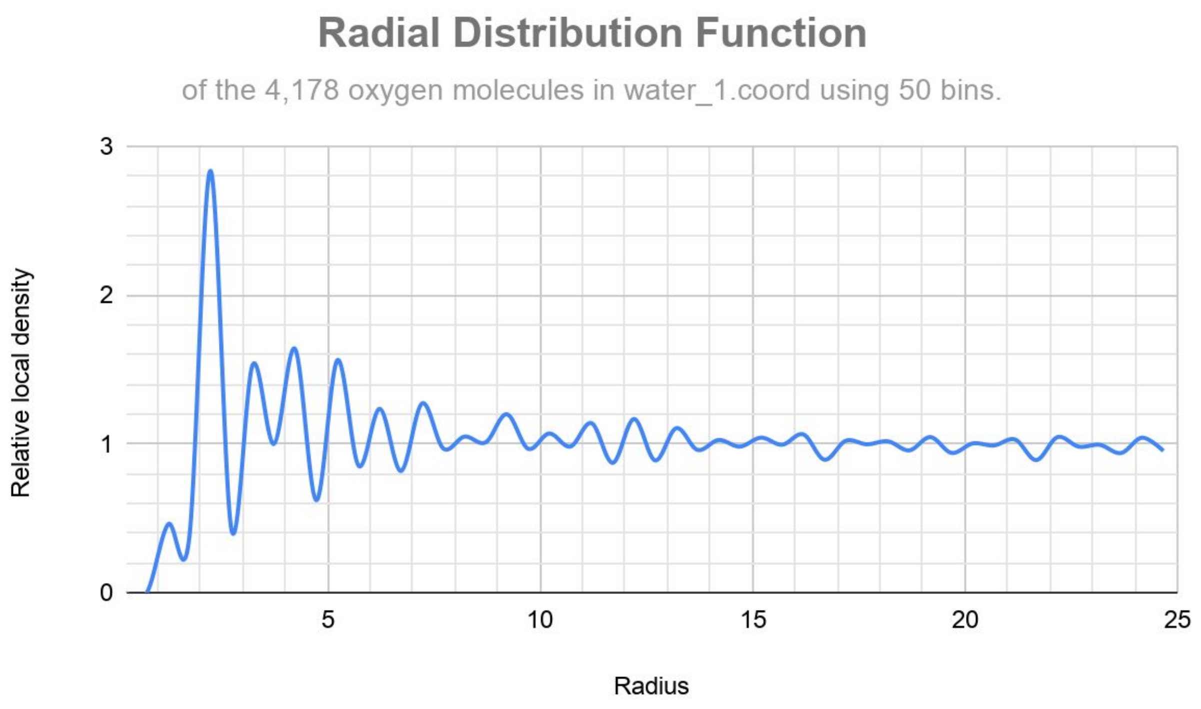
Time complexity

Conway’s Game of Life scales very well across multiple threads. For example, using two threads yields a near-perfect speedup of just under 2x. Using eight threads offers a speedup of roughly 6x, as compared to a theoretical maximum of 8x. A straight line from the bottom-left corner to the top-right corner in the speedup graph below represents perfect parallelization. This program falls short of this, but this is inevitable since parallelization overhead can’t be removed entirely. I expect results would improve if a larger grid size had been used. The good multithreading performance can be explained as a consequence of how processing each grid square or region presents an entirely isolated sub-problem. No communication between threads is necessary during any given game iteration. Problems which are easily decomposed like this usually handle multithreading well.





[2] Radial Distribution Function



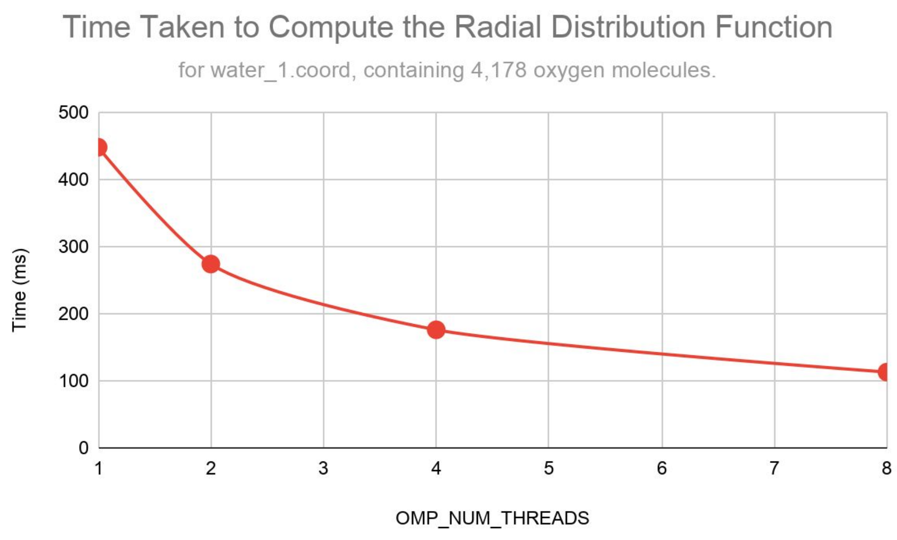
Source code

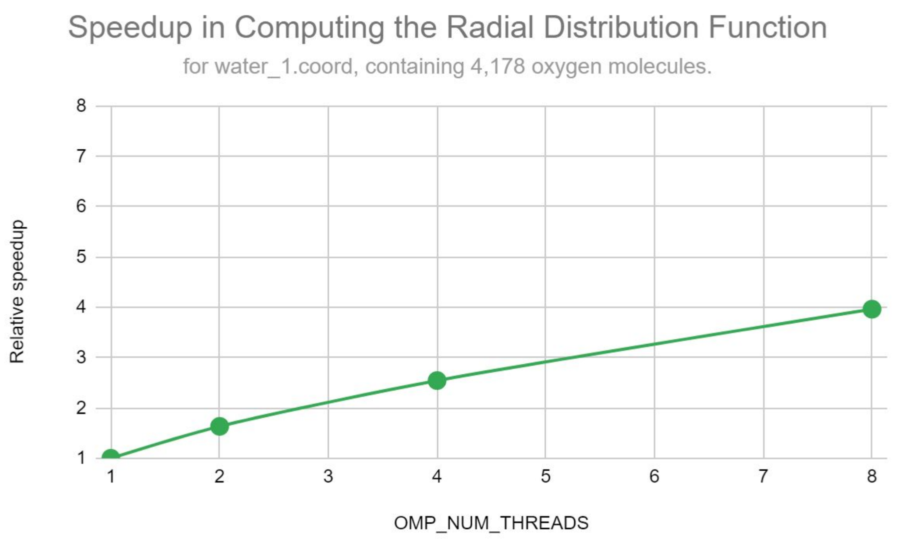
```c
// Radial distribution function calculator, using OpenMP.
// Written by Alec Dorrington.
// https://en.wikipedia.org/wiki/Radial_distribution_function

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <omp.h>

// Number of bins to divide radii into.
#define RESOLUTION 100
// Chunk size (number of molecules) used for dynamic decomposition.
#define CHUNKSIZE 10
// Name of file to write the RDF to.
#define DST_FILE "rdf.dat"

// Vector type, used for molecule coordinates.
struct vec_t { float x, y, z; };
```

```c
// Fluid type, containing molecules and cell size.
struct fluid_t {
  struct vec_t *molecules;
  int count;
  float width;
};

// Read fluid data (list of molecules) from a file.
struct fluid_t readFluid(char *fileName) {

  char line[100], name[10];
  struct vec_t pos;
  struct fluid_t fluid;
  int i, j = 0, n;
  FILE *file = fopen(fileName, "r");

  // Read number of molecules and cell size.
  fscanf(file, "%d\n", &n);
  fscanf(file, "%f", &(fluid.width));
  fluid.molecules = (struct vec_t*) malloc(n * sizeof(struct vec_t));
  fluid.count = 0;

  // Read each molecule.
  for(i = 0; i < n; i++) {
    fscanf(file, "%s %f %f %f", &name, &(pos.x), &(pos.y), &(pos.z));
    // Only include oxygen molecules.
    if(!strcmp(name, "O2")) fluid.molecules[fluid.count++] = pos;
  }
  fclose(file);
  return fluid;
}

// Write the radial distribution function to file as a list of bins.
void writeRdf(float *bins, struct fluid_t fluid, int res, char* fileName) {

  int i;
  float dr = fluid.width / 2.0F / res;
  FILE *file = fopen(fileName, "w");

  // Write each bin to the file.
  for(i = 0; i < res; i++) {
    fprintf(file, "%f %f\n", dr * (i + 0.5F), bins[i]);
  }
  fclose(file);
}

// Determine the minimum Euclidean distance between two vertices.
// Assumes periodic boundary conditions with the given cell width.
float distance(struct vec_t pos1, struct vec_t pos2, float width) {
  float xd, yd, zd;

  xd = abs(pos2.x - pos1.x); if(xd > width / 2.0F) xd = width - xd;
  yd = abs(pos2.y - pos1.y); if(yd > width / 2.0F) yd = width - yd;
  zd = abs(pos2.z - pos1.z); if(zd > width / 2.0F) zd = width - zd;

  return (float) sqrt(xd*xd + yd*yd + zd*zd);
}

// Normalize the RDF as a proportion of bulk density.
void normalize(float *bins, struct fluid_t fluid, int res) {

  int i;
  float r, dv;
  // Calculate radius step size and bulk density.
  float dr = fluid.width / 2.0F / res;
  float p = fluid.count / (float) pow(fluid.width, 3);

  // For each bin.
  for(i = 0; i < res; i++) {
    // Calculate the radius and change in volume.
    r = dr * (i+1);
    dv = 4.0F * (float) M_PI * r * r * dr;
    // Average over each choice of 'centre' particle, divide by
    // volume for local density, and normalize using bulk density.
    bins[i] /= fluid.count * dv * p;
  }
}

// Calculate the RDF of a fluid using some number of bins.
float* rdf(struct fluid_t fluid, int res) {

  int i, j, bin;
  float dist;
  float *p_bins, *bins = (float*) calloc(res, sizeof(float));

  #pragma omp parallel \
    shared(fluid, res, bins) private(i, j, dist, bin, p_bins)
  {
    p_bins = (float*) calloc(res, sizeof(float));

    // For each pair of particles, in parallel.
    #pragma omp for schedule(dynamic, CHUNKSIZE)
    for(i = 0; i < fluid.count; i++) {
      for(j = i+1; j < fluid.count; j++) {

        // Calculate the Euclidean distance between the particles.
        dist = distance(fluid.molecules[i], fluid.molecules[j], fluid.width);

        // If this distance is within the range considered.
        if(dist < fluid.width / 2.0F) {
          // Increment the bin corresponding to the distance.
          bin = (int) (res * dist / (fluid.width / 2.0F));
          p_bins[bin] += 2.0F;
        }
      }
    }
    // Merge the cumulative RDF bins of each thread.
    #pragma omp critical
    for(i = 0; i < res; i++) bins[i] += p_bins[i];
    free(p_bins);
  }
  // Normalize the results.
  normalize(bins, fluid, res);
  return bins;
}

// Get the number of milliseconds since the Unix epoch.
long timeMs() {

  struct timeval time;
  gettimeofday(&time, NULL);
  return time.tv_sec * 1000 + time.tv_usec / 1000;
}

int main(int argc, char **argv) {

  struct fluid_t fluid;
  float *bins;
  long t;

  if(argc != 2) printf("Usage: ./rdf [src file]\n");
  else {

    // Read the fluid and calculate the RDF.
    fluid = readFluid(argv[1]);
    t = timeMs();
    bins = rdf(fluid, RESOLUTION);

    // Write the RDF to DST_FILE.
    printf("Calculated the radial distribution function of a ");
    printf("%d-oxygen system in %dms.\n", fluid.count, timeMs() - t);
    writeRdf(bins, fluid, RESOLUTION, DST_FILE);

    free(fluid.molecules);
    free(bins);
  }
  return 0;
}
```

Time complexity

Computing the RDF handled multithreading worse than the Game of Life. Using two threads only offered a 1.6x speedup, as compared to a 2x maximum. The results are even worse when eight threads are used where a speedup of about 4x is achieved, which is only half of the 8x maximum. This is interesting, since, like with the Game of Life, this problem is seemingly very decomposable. That is, except for (a) the reduction at the end and (b) the normalization which follows. However, both of these computations only depend on the number of bins, not the number of molecules. Hence, I suspect using either less bins or more molecules would remedy this poor scaling behaviour. Another concern is how, in this case, and unlike that of the Game of Life, the problem sizes aren't equal (at least they aren't the way I implemented it). However, use of dynamic decomposition (which was also used for the previous task) should largely mitigate this concern.


Time Taken to Compute the Radial Distribution Function
for water_1.coord, containing 4,178 oxygen molecules.
Time (ms)
OMP_NUM_THREADS


Speedup in Computing the Radial Distribution Function
for water_1.coord, containing 4,178 oxygen molecules.
Relative speedup
OMP_NUM_THREADS