ECE 2230: Computer Systems Engineering
Machine Problem 1

Spring 2021

Due: 11:59 pm, Wednesday, January 27

1 Introduction

The goal of this machine problem is to familiarize you with the basics of maintaining information with dynamic memory allocation and pointers. This assignment provides a simple example of data abstraction for organizing information in an ordered list. We will develop a simple abstraction with a few key interfaces and a simple underlying implementation using sequential arrays. In a subsequent programming assignment, we will expand upon the interfaces and explore alternative implementations. We will refer to this abstract data type (ADT) as a sequential list.

2 Problem Statement

Molecular dynamics (MD) codes simulate the physical movement and interactions of molecules and atoms. MD is used in many problem areas (e.g., material science, biology, chemistry). MD is computationally intensive, and is often run in parallel on large-scale clusters or supercomputers.

In parallel environments, the physical domain simulated is decomposed spatially. In 2D, the domain is partitioned into squares similar to a checkerboard. In 3D, the domain is partitioned into cubes as shown in Figure 2. A single parallel process is responsible for atoms in each decomposed region. Over time, atoms move around and can migrate into a spatial region belonging to another parallel process. When this occurs, ownership of that atom switches to the process that manages that spatial region (i.e., that atom's data is sent over the network to the new owner and the current owner removes it from their list of atoms). For this project, we will write the ADT to store and operate on an array of atoms in a similar fashion to a MD simulation. Note that the `atom_t` structure contains vectors of length `NDIM` to accommodate the spatial components of that variable.

You are to write a C program that must consist of the following two files:

atom_list.c – contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.

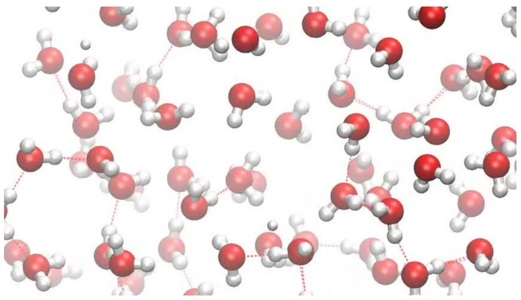

Figure 1: MD simulation of water molecules.

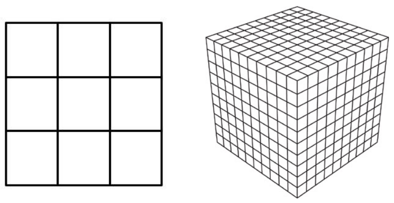
(a) 2D Domain (b) 3D Domain.

Figure 2: Domain decomposition for a MD simulation.

atom_list.h – contains the interface declarations for the ADT (e.g., constants, structure definitions, and prototypes). This file is provided and no changes can be made to it.

We provide a driver to test these codes. **DO NOT** modify the functionality of this file.

lab1.c – contains the `main()` function, menu code for handling simple input and output used to test our ADT, and any other functions that are not part of the ADT.

Your program must use an array of pointers to C structures that contain information for each atom. The atom information that is stored is represented with a C structure. The details are provided in the `atom_list.h` file, and no changes to this file are permitted. The data structure for the list is defined as follows:

```
struct atom_list_t {
int atom_count;        // current number of records in list
int atom_size;         // size of the list
struct atom_t **atom_ptr;
};

struct atom_t {
unsigned int atom_id;      // Unique atom id
unsigned int atomic_num;   // Atomic number for atom

float mass; // Atom's mass
float position[NDIM];      // Atom position
float momenta[NDIM];       // Atom momenta vector
float force[NDIM];          // Forces
float potential_energy;    // Potential energy per atom

};
```

The sequential list ADT must have the following interface:

```
struct atom_list_t *atom_list_construct(int size);
void atom_list_destruct(struct atom_list_t *);
int atom_list_number_entries(struct atom_list_t *);
int atom_list_add(struct atom_list_t *, struct atom_t *);
struct atom_t *atom_list_access(struct atom_list_t *, int index);
struct atom_t *atom_list_remove(struct atom_list_t *, int index);

void atom_list_advance_momenta(struct atom_list_t *, float dt);
void atom_list_advance_position(struct atom_list_t *, float dt);
void atom_list_compute_forces(struct atom_list_t *);


int atom_list_lookup_max_potential_energy(struct atom_list_t *, int potential_energy);
int atom_list_determine_inside_box(struct atom_t *, float x_min, float x_max, float y_min, float y_max)
struct atom_list_t *atom_list_form_migrate_list(struct atom_list_t *, float x_min, float x_max, float y
```

atom_list_construct should return a pointer to the header block for the data structure. The data structure includes an array of pointers where the size of the array is equal to the value passed in to the function (the size is a command line parameter as shown in `lab1.c`. See `list_size` in `lab1.c`). Each element in the array is defined as a pointer to a structure of type `atom_t`. Each pointer in the array should be initialized to `NULL`.

atom_list_destruct should free the array of type `atom_t *` (do not delete the atoms as they will be freed in main since it allocated them), and finally free the memory block of type `atom_list_t`.

atom_list_number_entries returns the current length of the atom list.

atom_list_add should take an `atom_t` memory block (that is already populated with input information) and insert it at the end of the list such that the list is sequential with no empty gaps between entries in the list. That is, the first record must be found at index position 0, the next ordered record at index position 1, etc. The function should return 1 if you inserted the new record into the list, or it should return -1 if the list is full and the insertion fails.

atom_list_access should return a pointer to the `atom_t` memory block that is found in the list index position specified as a parameter. If the index is out of bounds or if no atom record is found at the index position, the function returns `NULL`.

atom_list_remove should remove the memory block from the list and return the pointer to the memory block. The resulting list should still be sequential with no gaps between entries in the list. If the index given to the remove function is not valid, the function returns `NULL`.

atom_list_advance_momenta should update all the `atom_t` momenta vector components based on the formula: $momenta = momenta + \Delta t * force$.

atom_list_advance_position should update all the `atom_t` position vector components based on the formula: $position = position + \frac{\Delta t * momenta}{mass}$.

atom_list_compute_forces should assign new forces and a new potential energy for each atom by assigning a random number generated with `drand48()` to each of the fields. You will call `drand48()` each assignment.

atom_list_lookup_max_potential_energy should find the `atom_t` memory block at the lowest index in the list that matches the specified `potential_energy` and return the index position of the record within the list. If the `potential_energy` is not found, then return -1.

atom_list_determine_inside_box should return 1 if the atom passed to the function is positioned within the box given by the points (`x_min, y_min`) (`x_max, y_max`). If the atom is positioned outside the box, return 0. Consider any atom on the line of the bounding box as inside (i.e. vollyball rules).

atom_list_form_migrate_list should return a new `atom_list_t` that contains all atoms outside the box given by the points (`x_min, y_min`) (`x_max, y_max`). You can make the list's maximal size the same as the maximal size of the main atom list. If an atom is found to be outside the box, remove it from the main list and add it to the migrate list. If no atoms are outside the box, return the empty migrate list.

The driver file `lab1.c` provides the framework for input and output and testing the sequential list of atom list information. The code reads from standard input the commands listed below. The driver contains prints to standard output, and will be used when grading your code. The driver code in `lab1.c` calls functions found in atom_list.c. Based on the return information you will call the appropriate print statements. **DO NOT INCLUDE `printf` statements in your final submission of `atom_list.c`.** Doing so may impact grading as we look at the accuracy of standard output test cases. Two example input files and the exact output that is required are provided. Do not submit your code if your program cannot produce an exact match to the given output files. If your code is not a perfect match you must contact the instructor or TA and fix your code. These are the input commands:

INSERT
FIND potential_energy
REMOVE potential_energy
UPDATE dt
MIGRATE
PRINT
STATS
QUIT

The INSERT command allocates a dynamic memory block for the `atom_t` structure using `malloc()` and then calls the `fill_atom_record` function to prompt for each field of the record. After all the information is collected, an attempt to add the record to the list is made. The `atom_list_add` function can insert or reject the record, and prints a corresponding output message. The FIND command prints the information for the first atom record for which the `potential_energy` matches the `potential_energy` of an atom in the list. The REMOVE command removes the first entry of the list that matches the provided `potential_energy`, but also removes the record from the list (and frees the memory for the record). The UPDATE command emulates advancing the simulation through time by Δt (the amount of time to advance the simulation) calculating forces, updating momenta, and updating position. The MIGRATE command simulates the migration of atoms to another process. The STATS command prints the number of records in the list. The PRINT command prints each record if there are one or more records in the list. Finally, the QUIT command frees all the dynamic memory and ends the program.

3 Notes

1. The 12 `atom_list_*` function prototypes must be listed in `atom_list.h` and the corresponding functions must be found in the `atom_list.c` file. Code in `lab1.c` calls functions defined in `atom_list.c` only if its prototype is listed in `atom_list.h`. You can also add other "private" functions to `atom_list.c`, however, these private functions can only be called from within other functions in `atom_list.c`. The prototypes for your private functions **cannot** be listed in `atom_list.h`. Note we are using the principle of information hiding: code in `lab1.c` does not "see" any of the details of the data structure used in `atom_list.c`. The only information that `lab1.c` has about the atom list data structure is found in `atom_list.h` (and any "private" functions you add to `atom_list.c` are not available to `lab1.c`). The fact that `atom_list.c` uses an array of pointers is unimportant to `lab1.c`, and if we redesign the data structure no changes are required in `lab1.c` (including PRINT).

2. Recall that you compile your code using: `gcc -Wall -g lab1.c atom_list.c -o lab1` or `make`
   You can pipe my example test scripts as input using <. Collect output in a file using > For example, to run do `./lab1 10 < testinput.txt > testoutput.txt` The code you submit must compile using the `-Wall` flag and no compiler errors or warnings should be printed. (Windows and OS X users must verify that there are no warnings on a machine running Ubuntu.)
   An example `testinput.txt` and `expectedoutput.txt` files are provided. When you run your code on the `testinput.txt` file, your output must be identical to the file `expectedoutput.txt`. You can verify this using `diff testoutput.txt expectedoutput.txt`
   However, **the tests in `testinput.txt` are incomplete!** You must develop more thorough tests (e.g., attempt to delete from an empty list, or insert into a list that is already full). If you have access to a graphical display you can use `meld` in place of `diff`.

3. Be sure that your program does not have any memory leaks. That is, all dynamically allocated memory must be freed before the program ends. We will test for memory leaks with `valgrind`. You execute valgrind using:
   `valgrind --leak-check=yes ./lab1 10 < testinput.txt`
   The last line of output from valgrind must be: `ERROR SUMMARY: 0 errors from 0 contexts (suppressed: x from y)` You can ignore the values x and y because suppressed errors are not important and are hidden from you. In addition the summary of the memory heap must show: `All heap blocks were freed -- no leaks are possible`

4. Compress .c and .h files along with the test plan and test log into a ZIP file, and submit the ZIP file to Canvas by the deadline. **Your last submission is the one that will be graded.**

Work must be completed by each individual student. See the course syllabus for additional policies.