CISC 372: Parallel Computing

University of Delaware, Spring 2022

HW 2

This assignment is due Wednesday, Feb. 23, at noon (12:00 PM). Topics: sequential C, diffusion.

1. Instructions

From now on, I will assume that you have already checked out a local copy of the class repository (`372-2022S`) and your personal repository (`372-USER`). Furthermore, I will assume these two local copies are in the same directory (for example, your home directory). This is important because some of the programs you will write in your personal repository will need to use libraries that are in the class repository and they will therefore need to know where to find the class repository.

First, update the local copies. Do this often, since documents will be added to them continually throughout the semester.

Next, create a directory named `hw02` in the `hw` directory of your personal repository and add it to version control.

From now on, I will stop saying "and add to version control," because any source file or other file asked for in a homework assignment should obviously be added to version control (else how could we grade it?). But be sure to *not* add any *generated* files, such as anything generated by the compiler, i.e., no executable files, no object files, no output, etc., unless the instructions explicitly ask for such a thing.

Periodically, you should also commit the work in your personal repository. It is best to execute `svn commit` from the root directory of your local copy. Do this early and often, to make sure you don't lose work, and also because you might want to work on different machines which must be kept in sync.

(You can control which editor is used by Subversion and other Unix programs by setting the environment variable `EDITOR` to your favorite editor; for example

```
export EDITOR=emacs
```

will ensure that Subversion will use emacs for all of your commits. Put that line in your `.bash_profile` file in your home directory and it will be run every time you log on.)

If you have a short log message and don't want or need to go through an editor to enter your log message, you can use the `-m` option:

```
svn commit -m "Added a Makefile and fixed a small bug."
```

To summarize: you only have to `svn add` a file once. Once it has been added, every time you execute `svn commit`, the changes made to that file will be sent back to the central repository.

Remember: if you don't add it, and commit it, you haven't submitted it. And be sure to commit your final work. It is your last commit before the deadline that will "count" when we grade.

2. Problem 1: C types

Create a subdirectory of `hw02` called `types` (and, of course, add it to version control). Answer the following questions in a plain text file named `README` within the `types` directory.

Consider the following declarations. In each case, write the English name of the type of `x`.

Example: `double *x;`

Answer: pointer-to-double

(a) `int *x[];`
(b) `int (*x)[];`
(c) `double **x;`
(d) `unsigned long int (*x[])[n];`

In each of the following code snippets, data is copied from `x` to `y`. How many bytes of data are copied? Your answer should be a C expression.

Example:
```
int x[10], y[10];
...
for (int i=0; i<10; i++)
  x[i] = y[i];
```

Answer: `10*sizeof(int)`

Example: `int *x, *y; ...; y = x;`

Answer: `sizeof(int*)`

(e) `int x=10, y=x;`
(f) `int x[10]; int *y = x;`
(g)
```
int x[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
void f(int y[5]) { ... }
int main() { f(x); }
```
(h)
```
int x[10], y[10];
...
memcpy(y, x, 5*sizeof(int));
```

Your `README` file should contain just the answers, and look like this:

```
(a) this-is-the-name-of-the-type
(b) this-is-the-name-of-the-second-type
(c) yet-another-type
...
(h) ...
```

Nothing else should go in the file.

Grading rubric: 16 points: 2 points for each part.

3. Problem 2: reverse

Create a directory called `reverse` inside `hw02`. In a program named `reverse.c`, write a function

```
char * make_reverse(char * word) { ... }
```

which consumes a string, e.g., "`pool`" and returns a new string with the characters appearing in the reversed order, e.g., "`loop`".

Add a `main` function which takes any number (including 0) of command line arguments. It creates a new array whose entries are the results of calling `make_reverse` on the given arguments. It then prints the words of the new array, one per line. Finally, it frees any allocated memory.

Example:

```
> cc -o reverse.exec reverse.c
> ./reverse.exec pool ice Bach
loop
eci
hcaB
```

A few pointers:

(1) Remember that a string in C always ends with the null character `'\0'`.
(2) You may use the standard library function `strlen` by including header file `string.h`.
(3) Keep in mind that `argc` is one more than the number of command line arguments (`argv[0]` is the name of the program).
(4) Arrays of length 0 are not legal in C.

Add a `Makefile` so that typing `make` compiles the program into the executable `reverse.exec`, and typing `make test` runs 3 tests. Typing `make clean` should remove any generated files.

EXTRA CREDIT if you can set up the `Makefile` so that it also checks the output of the tests is correct.

Grading rubric: 24 points: compiling without error (2pt), compiling without warnings on strictest settings (2pt), correct output (10pt), correct use of `malloc/free` (4pt), 3 correct rules in Makefile (6pt). Extra credit up to 4 points.

4. Problem 3: diff2d

See the program `372-2022S/code/src/seq/diff1d/diff1d.c`.

This program simulates the diffusion of heat through a metal rod. The rod starts off at 0 degrees C., except for a small section in the middle, which is kept at 100 degrees C. throughout the simulation.

Over time, the heat flows (diffuses) from the middle hot point out to the ends of the rod. The ends of the rod are perfectly insulated so that no heat is ever lost (or gained) from the environment.

The mathematical model describing how the heat flows over time is a differential equation called the *heat equation.* In the computational model, we divide the rod into `n` discrete pixels and assign a temperature to each pixel. We then loop over discrete time steps, updating the pixel temperatures at each step.

In `diff1d.c`, the temperatures are stored in an array of doubles `u` of length `n`. The middle 4 cells are held at 100.0. The other interior cells are updated using the formula

```
  u_new[i] =  u[i] + k*(u[i+1] + u[i-1] - 2*u[i]);
```

where `k` is some constant that determines how fast heat moves through the rod. (The larger `k`, the faster heat flows. Note `k` must satisfy $0 <$ `k` < 0.5.)

Where does this formula come from? The idea is that heat flows into cell i from its two neighboring cells. The key physical principle is that the rate at which heat flows is proportional to the difference in temperature between cell i and its neighbor. So the new temperature of cell i (u'_i) is the old temperature of cell i (u_i) plus the contribution from the right neighbor, plus the contribution from the left neighbor:

$$\begin{aligned} u'_i &= u_i + k(u_{i+1} - u_i) + k(u_{i-1} - u_i) \\ &= u_i + k(u_{i+1} + u_{i-1} - 2u_i). \end{aligned}$$

The two boundary cells are updated using simpler formulas since there is only one neighbor:

$$\begin{aligned} u'_0 &= u_0 + k(u_1 - u_0) \\ u'_{n-1} &= u_{n-1} + k(u_{n-2} - u_{n-1}). \end{aligned}$$

`u_new` is also an array of length `n`, used to store the pixel temperatures for the new time step (i.e., what we write as u' in the math equations). It is necessary to have two copies of the array because you cannot modify the array while you are reading it. (If you modify `u[1]`, then the computation for `u[2]` will be wrong, because it requires the old value of `u[1]`.)

After `u_new` is computed, the pointers `u` and `u_new` are swapped. Now `u` points to the new values, and `u_new` points to the "old" values, which will get over-written on the next time step. The pointer-swapping approach is *much* faster than copying all `n` values from one array to the other.

The program uses the ANIM library to write its output. ANIM is a simple library for creating animated videos from simulations. The interface can be found in `372-2022S/code/include/anim.h`. You should familiarize yourself with this API, since you will use it for several problems throughout this course.

The code in `diff1d.c`

```
 af = ANIM_Create_heat(1,
                       (int[]){n},
                       (ANIM_range_t[]){{0,1}, {0,m}},
                       filename);
```

starts a new animation for a "heat graph":

```
/* Starts a new animation of the HEAT kind.

   dim: the dimension of the spatial domain (1, 2, or 3)
   lengths: number of pixels in each dimension; array of length dim
   ranges: the range of values represented by each axis in the spatial domain,
           plus one more range for the range of values taken on by the function
   filename: name of file to create, where data will be written
*/
ANIM_File ANIM_Create_heat(int dim, int lengths[dim],
                           ANIM_range_t ranges[dim+1],
                           char * filename);
```

In our case, the dimension is 1 (as a rod is 1-dimensional), the number of pixels is `n`, the rod will be measured in units ranging from 0.0 to 1.0, and the output values will take on numbers between 0.0 and 100.0. (The units on the rod, 0.0–1.0, are currently not used for anything; but in the future they may be used to label the points in the video.)

The code statement `ANIM_Write_frame(af, u)` causes the `n` doubles in `u` to be written to the ANIM file, adding a new frame to the animation.

Since diffusion simulations tend to change very slowly, instead of writing every time step to the file, we only write once every `wstep` ("write steps") time steps.

`ANIM_Status_update(stdout, nstep, i, &dots)` is used to print a textual "progress bar" to the terminal. This is convenient for long running computations, so the user knows progress is being made and approximately how much work remains.

`ANIM_Close(af)` is called after the last frame has been added, and closes the file.

The output of `diff1d.exec` is an anim file named `diff1d.anim`. This file can be examined from the shell by `anim2txt diff1d.anim`, which prints all the values. Type `anim2txt -h` to learn the different options for `anim2txt`; in particular, you can control exactly which values are printed.

There are also command line tools to covert an ANIM file to different graphical formats. `anim2gif` converts the ANIM file to an animated GIF; these can be viewed in a web browser or in many different media players. `anim2mp4` converts to MP4 (MPEG-4) video, which is the most popular video format and can be played by almost every media player (e.g., QuickTime on the Mac).

When you run `diff1d.exec`, or any other program that does significant computation, you want to be sure it executes on a back-end machine, not the login machine (`cisc372.cis.udel.edu`). This is accomplished using SLURM's `srun` command, e.g.,

```
srun -n 1 ./diff1d.exec
```

The "`-n 1`" indicates you will run a single process (a sequential program), and SLURM will allocate it a single core. The provided `Makefile` does this automatically as the variable `RUN` expands to `srun -n 1` on `cisc372.cis.udel.edu`.

If you run `diff1d.exec` on the login machine, the machine will grind to a near halt and no one will be able to get work done. Use `srun`.

Typing `make test` in the `diff1d` directory will run through the whole process, generating the MP4 video file.

You can transfer the MP4 file to your local computer by executing a command such as the following on your local computer:

```
scp USER@cisc372.cis.udel.edu:372-2022S/code/src/seq/diff1d/diff1d.mp4 .
```

Try it, and watch the video. Pure blue represents 0 deg., pure red represents 100 deg., and temperatures in between are represented by some shade on the blue-red spectrum.

Your job in this problem is to write a 2-dimensional version of diff1d.

Create a directory `diff2d` and call your program `diff2d.c`. It will simulate the diffusion of heat through a square metal plate. The plate starts off at 0 deg., but the 4×4 pixel square in the center is kept at 100 deg.

Now a cell in the interior has 4 neighbors: the upper, lower, left, and right neighbor. The update formula for an interior point (other than the center square) is therefore given by:

$$\begin{aligned} u'_{i,j} &= u_{i,j} + k((u_{i,j+1} - u_{i,j}) + (u_{i,j-1} - u_{i,j}) + (u_{i-1,j} - u_{i,j}) + (u_{i+1,j} - u_{i,j})) \\ &= u_{i,j} + k(u_{i,j+1} + u_{i,j-1} + u_{i-1,j} + u_{i+1,j} - 4u_{i,j}) \end{aligned}$$

The boundaries have to be updated separately using 8 different equations: 4 for each of the edges, and 4 for each corner. Here is the update formula for a cell on the top edge that is not on a corner:

$$\begin{aligned} u'_{i,n-1} &= u_{i,n-1} + k((u_{i,n-2} - u_{i,n-1}) + (u_{i-1,n-1} - u_{i,n-1}) + (u_{i+1,n-1} - u_{i,n-1})) \\ &= u_{i,n-1} + k(u_{i,n-2} + u_{i-1,n-1} + u_{i+1,n-1} - 3u_{i,n-1}) \end{aligned}$$

Here is the update formula for a cell in the top left corner:

$$\begin{aligned} u'_{0,n-1} &= u_{0,n-1} + k((u_{0,n-2} - u_{0,n-1}) + (u_{1,n-1} - u_{0,n-1})) \\ &= u_{0,n-1} + k(u_{0,n-2} + u_{1,n-1} - 2u_{0,n-1}) \end{aligned}$$

A `Makefile` and starter `diff2d.c` are provided for you; copy these over to your new `diff2d` directory, and add them to version control.

For testing, I suggest you start with very small values of `n`, and scale up only after you're confident you got all the bugs out.

Grading rubric: 30 points: program is present and compiles without error (1pt); compiles without warnings (2pt); `Makefile` present and works correctly (1pt); no generated files committed (2pt); correct global declarations (4pt); correct setup (4pt); correct teardown (2pt); correct update (6pt); correct main (2pt); generated output video looks visually plausible, showing heat diffusing from center towards edges (6pt).