CS 475/575 -- Spring Quarter 2021

Project #3

Functional Decomposition

100 Points

Due: May 5

This page was last updated: April 19, 2021

Introduction

This project will use parallelism, not for speeding data-computation, but for programming-convenience. You will create a month-by-month simulation in which each agent of the simulation will execute in its own thread where it just has to look at the state of the world around it and react to it.

You will also get to exercise your creativity by adding an additional "agent" to the simulation, one that impacts the state of the other agents and is impacted by them.

Requirements

1. You are creating a month-by-month simulation of a grain-growing operation. The amount the grain grows is affected by the temperature, amount of precipitation, and the number of deer around to eat it. The number of deer depends on the amount of grain available to eat.

2. The "state" of the system consists of the following global variables:

```
int     NowYear;            // 2021 - 2026
int     NowMonth;           // 0 - 11

float   NowPrecip;          // inches of rain per month
float   NowTemp;            // temperature this month
float   NowHeight;          // grain height in inches
int     NowNumDeer;         // number of deer in the current population
```

3. Your basic time step will be one month. Interesting parameters that you need are:

```
const float GRAIN_GROWS_PER_MONTH =         9.0;
const float ONE_DEER_EATS_PER_MONTH =       1.0;

const float AVG_PRECIP_PER_MONTH =          7.0;    // average
const float AMP_PRECIP_PER_MONTH =          6.0;    // plus or minus
const float RANDOM_PRECIP =                 2.0;    // plus or minus noise
```

```
const float AVG_TEMP =                          60.0;   // average
const float AMP_TEMP =                          20.0;   // plus or minus
const float RANDOM_TEMP =                       10.0;   // plus or minus noise

const float MIDTEMP =                           40.0;
const float MIDPRECIP =                         10.0;
```

Units of grain growth are inches.
Units of temperature are degrees Fahrenheit (°F).
Units of precipitation are inches.

4. Because you know ahead of time how many threads you will need (3 or 4), start the threads with a `parallel sections` directive:

```
omp_set_num_threads( 4 );       // same as # of sections
#pragma omp parallel sections
{
        #pragma omp section
        {
                Deer( );
        }

        #pragma omp section
        {
                Grain( );
        }

        #pragma omp section
        {
                Watcher( );
        }

        #pragma omp section
        {
                MyAgent( );     // your own
        }
}       // implied barrier -- all functions must return in order
        // to allow any of them to get past here
```

5. Put this at the top of your program to make it a global:
```
unsigned int seed = 0;
```

6. The temperature and precipitation are a function of the particular month:

```
float ang = (  30.*(float)NowMonth + 15.  ) * ( M_PI / 180. );

float temp = AVG_TEMP - AMP_TEMP * cos( ang );
NowTemp = temp + Ranf( &seed, -RANDOM_TEMP, RANDOM_TEMP );

float precip = AVG_PRECIP_PER_MONTH + AMP_PRECIP_PER_MONTH * sin( ang );
NowPrecip = precip + Ranf( &seed,  -RANDOM_PRECIP, RANDOM_PRECIP );
if( NowPrecip < 0. )
        NowPrecip = 0.;
```

To keep this simple, a year consists of 12 months of 30 days each. The first day of winter is considered to be January 1. As you can see, the temperature and precipitation follow cosine and sine wave patterns with some randomness added.

7. Starting values are:

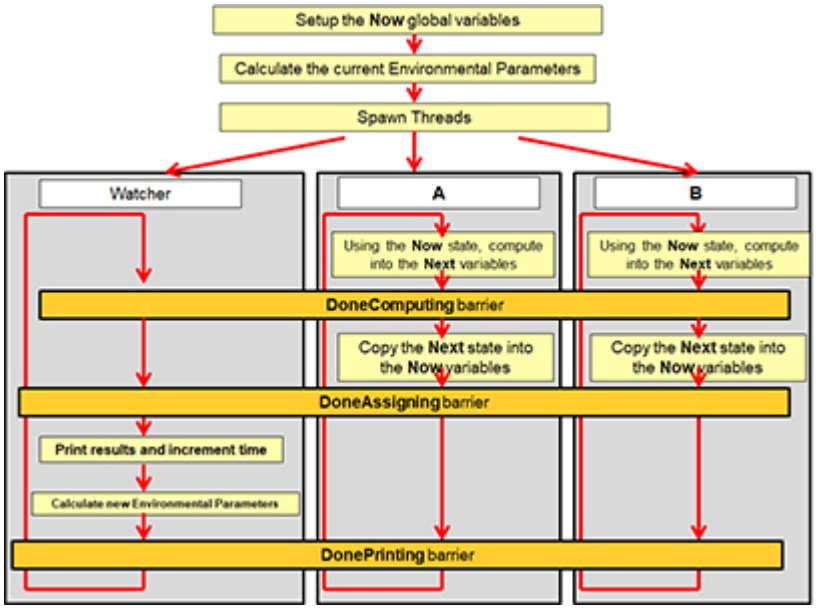
```
// starting date and time:
NowMonth =    0;
NowYear  = 2021;

// starting state (feel free to change this if you want):
NowNumDeer = 1;
NowHeight =  1.;
```

8. **In addition to this, you must add in some other phenomenon that directly or indirectly controls the growth of the grain and/or the deer population.** Your choice of this is up to you.

9. You are free to tweak the constants to make everything turn out "more interesting".

Use of Threads

As shown here, you will spawn three threads (four, when you add your own agent):



The **GrainGrowth** and **Deer** threads will each compute the next grain height and the next number of deer based on the current set of global state variables. They will compute these into local, temporary, variables. They both then will hit the **DoneComputing** barrier.

At that point, both of those threads are done computing using the current set of global state variables. Each thread should then copy the local variable into the global version. All 3 threads will then hit the **DoneAssigning** barrier.

At this point, the Watcher thread will print the current set of global state variables, increment the month count, and then use the new month to compute the new Temperature and Precipitation. Note that the **GrainGrowth** and **Deer** threads can't proceed because there is a chance they would re-compute the global state variables before they are done being printed. All 3 threads will then hit the **DonePrinting** barrier.

After spawning the threads, the main program should wait for the **parallel sections** to finish.

Each thread should return when the year hits 2027 (giving us 6 years, or 72 months, of simulation).

Remember that this description is for the core part of the project, before you add your own agent to the simulation. That will involve another thread and some additional interaction among the global state variables.

Quantity Interactions

The Carrying Capacity of the deer is the number of inches of height of the grain. If the number of deer exceeds this value at the end of a month, decrease the number of deer by one. If the number of deer is less than this value at the end of a month, increase the number of deer by one.

Each month you will need to figure out how much the grain grows. If conditions are good, it will grow by `GRAIN_GROWS_PER_MONTH`. If conditions are not good, it won't.

You know how good conditions are by seeing how close you are to an ideal temperature (°F) and precipitation (inches). Do this by computing a Temperature Factor and a Precipitation Factor like this:

$$TF = e^{-\left(\frac{T - MidTemp}{10}\right)^2}$$

$$PF = e^{-\left(\frac{P - MidPrecip}{10}\right)^2}$$

Note that there is a standard math function, exp(x), to compute e-to-the-x:

```
float tempFactor = exp(   -SQR(  ( NowTemp - MIDTEMP ) / 10.  )   );

float precipFactor = exp(   -SQR(  ( NowPrecip - MIDPRECIP ) / 10.  )   );
```

I like squaring things with another function:

```
float
SQR( float x )
{
        return x*x;
}
```

You then use `tempFactor` and `precipFactor` like this:

```
 float nextHeight = NowHeight;
 nextHeight += tempFactor * precipFactor * GRAIN_GROWS_PER_MONTH;
 nextHeight -= (float)NowNumDeer * ONE_DEER_EATS_PER_MONTH;
```

Be sure to clamp `nextHeight` against zero, that is:

```
if( nextHeight < 0. ) nextHeight = 0.;
```

Something like this will work for the number of deer:

```
int nextNumDeer = NowNumDeer;
int carryingCapacity = (int)( NowHeight );
if( nextNumDeer < carryingCapacity )
        nextNumDeer++;
else
        if( nextNumDeer > carryingCapacity )
                nextNumDeer--;

if( nextNumDeer < 0 )
        nextNumDeer = 0;
```

Structure of the Simulation Functions

Each simulation function will have a structure that looks like this:

```
while( NowYear < 2027 )
{
	// compute a temporary next-value for this quantity
	// based on the current state of the simulation:
	. . .

	// DoneComputing barrier:
	#pragma omp barrier
	. . .

	// DoneAssigning barrier:
	#pragma omp barrier
	. . .

	// DonePrinting barrier:
	#pragma omp barrier
	. . .
}
```

Doing the Barriers on Visual Studio

You will probably get this error when doing this type of project in Visual Studio:

'#pragma omp barrier' improperly nested in a work-sharing construct

The OpenMP specifications says: "All threads in a team must execute the barrier region." Presumably this means that placing corresponding barriers in the different functions does not qualify, but somehow gcc/g++ are OK with it.

Here is a work-around. Instead of using the
#pragma omp barrier
line, use this:
WaitBarrier( );

You must call **InitBarrier(n)** as part of your setup process, where **n** is the number of threads you will be waiting for at this barrier. Presumably this is **3** before you add your own quantity and is **4** after you add your own quantity.

I'm not particularly proud of this code, but it seems to work. To make this happen, first declare the following global variables:

```
omp_lock_t	Lock;
int		NumInThreadTeam;
int		NumAtBarrier;
int		NumGone;
```

Here are the function prototypes:

```
void	InitBarrier( int );
void	WaitBarrier( );
```

Here is the function code:

```
// specify how many threads will be in the barrier:
```

```
//      (also init's the Lock)

void
InitBarrier( int n )
{
        NumInThreadTeam = n;
        NumAtBarrier = 0;
        omp_init_lock( &Lock );
}


// have the calling thread wait here until all the other threads catch up:

void
WaitBarrier( )
{
        omp_set_lock( &Lock );
        {
                NumAtBarrier++;
                if( NumAtBarrier == NumInThreadTeam )
                {
                        NumGone = 0;
                        NumAtBarrier = 0;
                        // let all other threads get back to what they were doing
                        // before this one unlocks, knowing that they might immediately
                        // call WaitBarrier( ) again:
                        while( NumGone != NumInThreadTeam-1 );
                        omp_unset_lock( &Lock );
                        return;
                }
        }
        omp_unset_lock( &Lock );

        while( NumAtBarrier != 0 );     // this waits for the nth thread to arrive

        #pragma omp atomic
        NumGone++;                      // this flags how many threads have returned
}
```

Random Numbers

How you generate the randomness is up to you. As an example (which you are free to use), Joe Parallel wrote a couple of functions that return a random number between a user-given low value and a high value (note that the name overloading is a C++-ism, not a C-ism):

Note that this is using the **reentrant** version of *rand*, called *rand_r*.
rand, like you used in Project #1, actually keeps internal state (the current seed), which you now know is a dangerous thing. This version is safer.

```
#include <stdlib.h>
unsigned int seed = 0;


float x = Ranf( &seed, -1.f, 1.f );

. . .

float
Ranf( unsigned int *seedp,  float low, float high )
{
        float r = (float) rand_r( seedp );              // 0 - RAND_MAX
```

```
        return(   low  +  r * ( high - low ) / (float)RAND_MAX   );
}


int
Ranf( unsigned int *seedp, int ilow, int ihigh )
{
        float low = (float)ilow;
        float high = (float)ihigh + 0.9999f;

        return (int)(  Ranf(seedp, low,high) );
}
```

Results

Turn in your code and your PDF writeup. Be sure your PDF is a file all by itself, that is, not part of any zip file. Your writeup will consist of:

1. What your own-choice quantity was and how it fits into the simulation.

2. A table showing values for temperature, precipitation, number of deer, height of the grain, and your own-choice quantity as a function of month number.

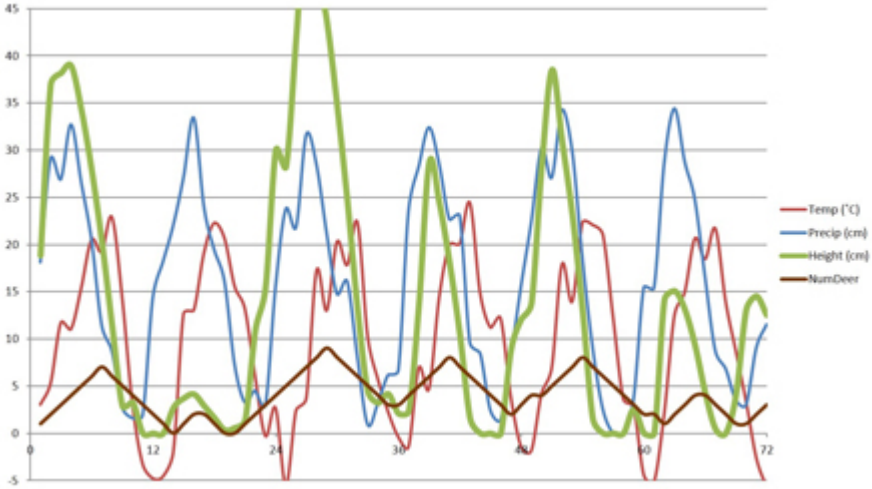
3. A graph showing temperature, precipitation, number of deer, height of the grain, and your own-choice quantity as a function of month number. Note: if you change the units to °C and centimeters, the quantities might fit better on the same set of axes.

   cm = inches * 2.54
   °C = (5./9.)*(°F-32)

   This will make your heights have larger numbers and your temperatures have smaller numbers.

4. A commentary about the patterns in the graph and why they turned out that way. What evidence in the curves proves that your own quantity is actually affecting the simulation correctly?

Example Output



Grading:

Feature	Points

Simulate grain growth and deer population	20
Simulate your own quantity	20
Table of Results	10
Graph of Results	20
Commentary	30
Potential Total	**100**