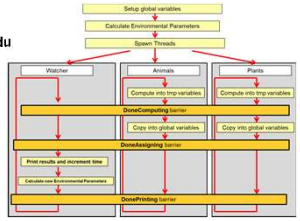
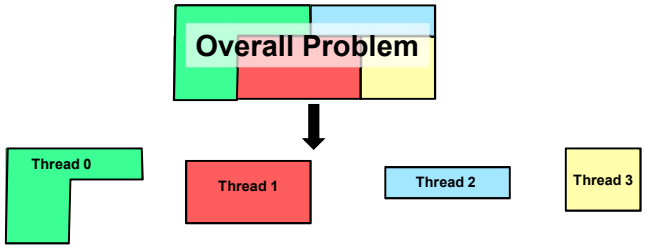
Functional (Task) Decomposition

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu



Oregon State University
Computer Graphics

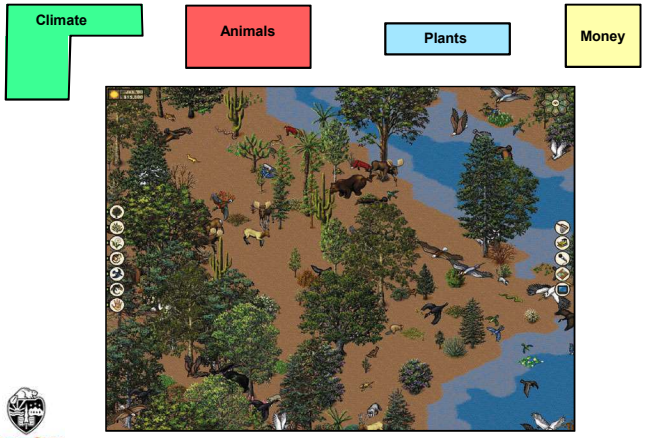
The Functional (or Task) Decomposition Design Pattern



A good example of this is the computer game *SimPark*.

Oregon State University
Computer Graphics

The Functional (or Task) Decomposition Design Pattern

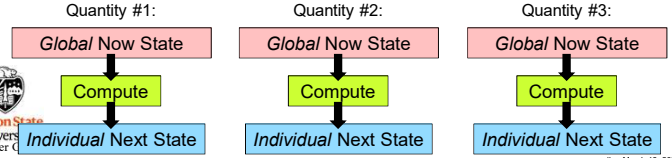


Credit: Maxis (Sim Park)

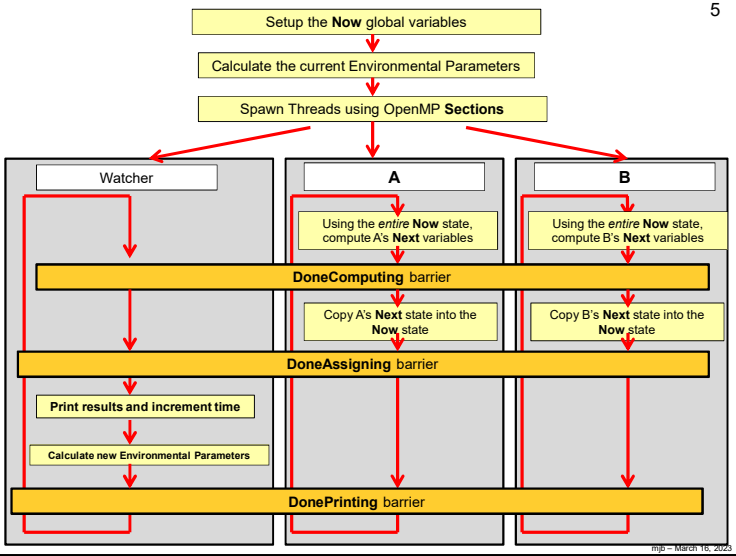
Oregon State University
Computer Graphics

How is this is different from Data Decomposition (such as the OpenMP for-loops)

- This is done less for performance and more for programming convenience.
- This is often done in simulations, where each quantity in the simulation needs to make decisions about what it does *next* based on what it and all the other global quantities are doing *right now*.
- Each quantity takes *all* of the "Now" state data and computes its own "Next" state.
- The biggest trick is to synchronize the different quantities so that each of them is seeing only what the others' data values are *right now*. Nobody is allowed to switch their data states until they are *all* done consuming the current data and thus are ready to switch together.
- The synchronization is accomplished with barriers.



Oregon State University
Computer Graphics



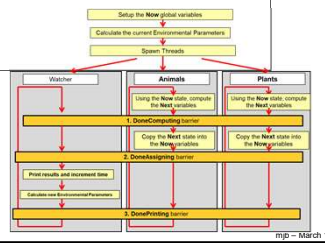
The Functional Decomposition Design Pattern

```
int
main( int argc, char *argv[ ] )
{
        . . .
        omp_set_num_threads( 3 );
        InitBarrier( 3 );                    // don't worry about this for now, we will get to this later

        #pragma omp parallel sections
        {
                #pragma omp section
                {
                        Watcher( );
                }

                #pragma omp section
                {
                        Animals( );
                }

                #pragma omp section
                {
                        Plants( );
                }
        }  // implied barrier -- all functions must return to get past here
}
```

Oregon State University
Computer Graphics

The Functional Decomposition Design Pattern

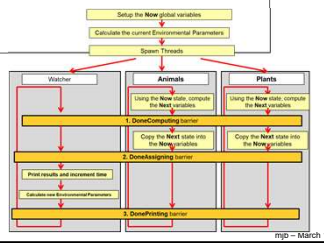
```
void
Watcher( )
{
        while( << You decide how to know when it's all finished? >> )
        {
                // do nothing
                WaitBarrier( );          // 1.

                // do nothing
                WaitBarrier( );          // 2.

                << write out the "Now" state of data >>

                << advance time and re-compute all environmental variables >>

                WaitBarrier( );          // 3.
        }
}
```



Oregon State University
Computer Graphics

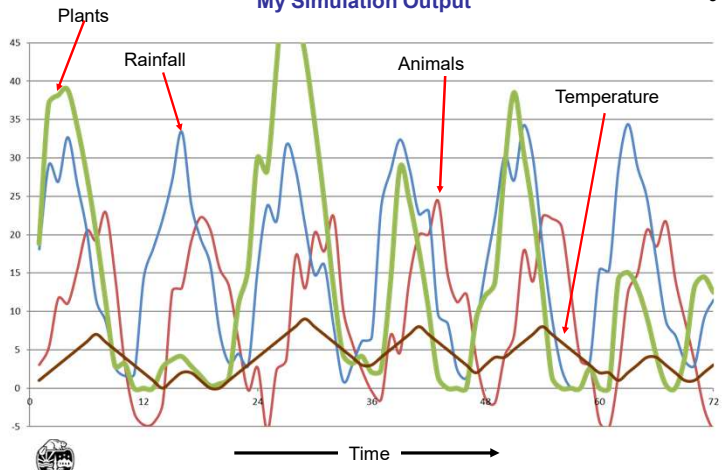
The Functional Decomposition Design Pattern

```
void
Animals( )
{
        while( << You decide how to know when it's all finished? >> )
        {
                int nextXXX= << function of what all states are right Now >>
                . . .
                WaitBarrier( );          // 1.

                NowXXX = nextXXX;        // copy the computed next state to the Now state

                WaitBarrier( );          // 2.

                // do nothing
                WaitBarrier( );          // 3.
        }
}
```



Oregon State University
Computer Graphics

My Simulation Output



Oregon State University
Computer Graphics

You Might Have to Make Your Own Barrier Function

Why can't we just use **#pragma omp barrier** ?

The Functional Decomposition is a good example of when you sometimes can't.

There are two ways to think about how to allow a program to implement a barrier:

1. Make a thread block at a specific location in the code. Keep blocking until all threads have blocked there.
2. Make a thread block when it asks to "Wait". Keep blocking until all threads have blocked by asking to "Wait".

- g++ apparently allows both #1 and #2
- Visual Studio *requires* #1
- The Functional Decomposition shown here wants to have #2, because the barriers need to be in different functions
- The OpenMP specification only allows for #1.

Oregon State University
Computer Graphics

Sometimes You Have to Make Your Own Barrier Function


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

void
InitBarrier( int n )
{
    NumInThreadTeam = n;                    // number of threads you want to block at the barrier
    NumAtBarrier = 0;
    omp_init_lock( &Lock );
}

void
WaitBarrier( )
{
    omp_set_lock( &Lock );
    {
        NumAtBarrier++;
        if( NumAtBarrier == NumInThreadTeam )          // release the waiting threads
        {
            NumGone = 0;
            NumAtBarrier = 0;
            // let all other threads return before this one unlocks:
            while( NumGone != NumInThreadTeam - 1 );
            omp_unset_lock( &Lock );
            return;
        }
    }
    omp_unset_lock( &Lock );

    while( NumAtBarrier != 0 );             // all threads wait here until the last one arrives …

    #pragma omp atomic                      // … and sets NumAtBarrier to 0
        NumGone++;
}
```

Oregon State University
Computer Graphics

The WaitAtBarrier( ) Logic

Thread #0	Thread #1	Thread #2	NumInThreadTeam	NumAtBarrier	NumGone
			3	0	
Calls WaitBarrier( )			3	0	
Sets the lock			3	0	
Increments NumAtBarrier			3	1	
NumAtBarrier != NumInThreadTeam			3	1	
Unsets the lock			3	1	
Stuck at while-loop #2			3	1	
	Calls WaitBarrier( )		3	1	
	Sets the lock		3	1	
	Increments NumAtBarrier		3	2	
	NumAtBarrier != NumInThreadTeam		3	2	
	Unsets the lock		3	2	
	Stuck at while-loop #2		3	2	
		Calls WaitBarrier( )	3	2	
		Sets the lock	3	2	
		Increments NumAtBarrier	3	3	
		NumAtBarrier == NumInThreadTeam	3	3	
		Sets NumGone	3	3	0
		Sets NumAtBarrier	3	0	0
		Stuck at while-loop #1	3	0	0
Falls through while-loop #2			3	0	0
Increments NumGone			3	0	1
Returns			3	0	1
	Falls through while-loop #2		3	0	2
	Increments NumGone		3	0	2
	Returns		3	0	2
		Falls through while-loop #1	3	0	2
		Unsets the lock	3	0	2
		Returns	3	0	2

Oregon State University
Computer Graphics