The Functional (or Task) Decomposition Design Pattern

A good example of this is the computer game SimPark.

How is this 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 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 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.

The Functional Decomposition Design Pattern

Credit: Maxis (Sim Park)

The Functional Decomposition Design Pattern

```
int main(int argc, char *argv[]) {
  omp_set_num_threads(3);
  InitBarrier(3);
  #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
  // don’t worry about this for now, we will get to this later
  omp_set_num_threads(3);
}
```
void Watcher( )
{
    while( << You decide how to know when 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.
    }
}

void Animals( )
{
    while( << You decide how to know when 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.
    }
}

The Functional Decomposition Design Pattern

The Functional Decomposition Design Pattern

My Simulation Output

You Might Have to Make Your Own Barrier Function

Sometimes You Have to Make Your Own Barrier Function

The WaitAtBarrier( ) Logic

The Functional Decomposition Design Pattern

The Functional Decomposition Design Pattern

You Might Have to Make Your Own Barrier Function

There are two ways to think about how to allow a program to use a barrier:
1. Let the barrier happen at a specific location in the code
2. Let the barrier work after a specific number of threads have gotten there

- g++ 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

Sometimes You Have to Make Your Own Barrier Function

omp_lock_t Lock;
int NumInThreadTeam;
int NumAtBarrier;
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++;
}