Functional (Task) Decomposition

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 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.
Setup the **Now** global variables

Calculate the current Environmental Parameters

Spawn Threads using OpenMP Sections

The Functional Decomposition Design Pattern

```c
int main( int argc, char *argv[] )
{
    ...  
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // do nothing
            WaitBarrier(); // 1.
            // do nothing
            WaitBarrier(); // 2.
            // advance time and re-compute all environmental variables >>
            WaitBarrier(); // 3.
            // do nothing
            WaitBarrier(); // 4.
            // copy the computed next state to the Now state
            NowXXX = nextXXX; // 5.
            // do nothing
            WaitBarrier(); // 6.
            ...  
    }
    // implied barrier -- all functions must return to get past here
}
```

The Functional Decomposition Design Pattern

```c
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.
    }
}
```

The Functional Decomposition Design Pattern

```c
void Animals( )
{
    while( << You decide how to know when it's all finished? >> )
    {
        int nextXXX << function of what all states are right Now >>
        ...  
        // do nothing
        WaitBarrier(); // 1.
        NowXXX = nextXXX; // copy the computed next state to the Now state
        WaitBarrier(); // 2.
        // do nothing
        WaitBarrier(); // 3.
    }
}
We Have to Make Our Own Barrier Function

Why can’t we just use #pragma omp barrier?

Functional Decomposition is a good example of when you can’t.

There are two ways to think about how to allow a program to implement a barrier:
1. Make a thread wait at a specific address in the code. Keep waiting until all threads are waiting there.
2. Make a thread wait when it specifically asks to “Wait”. Keep waiting until all threads have asked to “Wait”.

Both of these sound legitimate, but:
- The OpenMP specification only allows for #1.
- The Functional Decomposition described here wants to use #2, because the waiting needs to happen at different addresses in different functions.

```c
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_unsync_lock( &Lock );
                }
                return;
            }
    }
    omp_unset_lock( &Lock );
    while( NumAtBarrier != 0 ); // all threads wait here until the last one arrives …
    #pragma omp atomic
    NumGone++;
}
```

The WaitAtBarrier() Logic

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Thread KO</th>
<th>Thread KO_1</th>
<th>NumInThreadTeam</th>
<th>NumAtBarrier_1</th>
<th>NumGone_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Returns

- Args through worker loop
- Immune to NUMA
- Returns
- Args through worker loop
- Immune to NUMA
- Returns