Functional (Task) Decomposition

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

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*. 
The Functional (or Task) Decomposition Design Pattern

Credit: Maxis (Sim Park)
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**

**Watcher**

Using the *entire Now* state, compute A's **Next** variables

DoneComputing barrier

Copy A's **Next** state into the *Now* state

DoneAssigning barrier

Print results and increment time

Calculate new Environmental Parameters

DonePrinting barrier

Using the *entire Now* state, compute B's **Next** variables

Copy B's **Next** state into the *Now* state
The Functional Decomposition Design Pattern

```c
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
}
```
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.
    }
}
```
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.
    }
}

---

The Functional Decomposition Design Pattern
My Simulation Output

- Plants
- Rainfall
- Animals
- Temperature

Time

Oregon State University
Computer Graphics

mjb – March 16, 2023
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.
Sometimes You Have to Make Your Own Barrier Function

```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_unset_lock( &Lock );
                return;
            }
    }
    omp_unset_lock( &Lock );

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

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

## The WaitAtBarrier() Logic

<table>
<thead>
<tr>
<th>Thread #0</th>
<th>Thread #1</th>
<th>Thread #2</th>
<th>NumInThreadTeam</th>
<th>NumAtBarrier</th>
<th>NumGone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls WaitBarrier()</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Sets the lock</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Increments NumAtBarrier</td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>NumAtBarrier != NumInThreadTeam</td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Unsets the lock</td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stuck at while-loop #2</td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calls WaitBarrier()</td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sets the lock</td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Increments NumAtBarrier</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NumAtBarrier != NumInThreadTeam</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unsets the lock</td>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stuck at while-loop #2</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Falls through while-loop #2</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Increments NumGone</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Returns</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Falls through while-loop #2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Increments NumGone</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Falls through while-loop #1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unsets the lock</td>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Returns</td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>