The Functional (or Task) Decomposition Design Pattern

**Overall Problem**

- **Thread 0**
- **Thread 1**
- **Thread 2**
- **Thread 3**

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

- This is being done less for performance and more for programming convenience.
- This is often done in simulations, where each “chunk” of the simulation needs to make decisions about what it does next based on what it and the other chunks are doing right now.
- Each chunk takes all of “Now” state data and computes a “Next” state.
- The biggest trick is to synchronize the different chunks of the simulation so that each of them is seeing only what the others’ states are right now. Nobody can be allowed to switch their state to “next” until they are all ready to switch together.
- The synchronization is accomplished with **barriers**.

```c
omp_lock_t Lock;
int main(int argc, char *argv[])
{
    . . .
    omp_init_lock( &Lock );
    omp_set_num_threads( 3 );
    InitBarrier( 3 );
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            Watcher( );
        }
        #pragma omp section
        {
            Animals( );
        }
        #pragma omp section
        {
            Plants( );
        }
    }
    . . .
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            Print results and increment time
        }
    }
    . . .
}
```
void Animals() {
    while (< how to know when finished? >>) {
        int nextXXX = (function of all states);
        fprintf(stderr, "Animals waiting at #1.
"); WaitBarrier();
        fprintf(stderr, "Animals resuming at #1.
");
        NowXXX = nextXXX;
        fprintf(stderr, "Animals waiting at #2.
"); WaitBarrier();
        fprintf(stderr, "Animals resuming at #2.
");
        fprintf(stderr, "Animals waiting at #3.
"); WaitBarrier();
        fprintf(stderr, "Animals resuming at #3.
");
    }
}

void Watcher() {
    while (< how to know when finished? >>) {
        fprintf(stderr, "Watcher waiting at #1.
"); WaitBarrier();
        fprintf(stderr, "Watcher resuming at #1.
");
        fprintf(stderr, "Watcher waiting at #2.
");
        fprintf(stderr, "Watcher resuming at #2.
");
        << write out "Now" state of data >>
        << advance time and re-compute all environmental variables >>
        fprintf(stderr, "Watcher waiting at #3.
"); WaitBarrier();
        fprintf(stderr, "Watcher resuming at #3.
");
    }
}

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

void WaitBarrier() {
    omp_set_lock(&Lock);
    {
        NumAtBarrier++;
        if (NumAtBarrier == NumInThreadTeam) // release 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 until the last one arrives
    #pragma omp atomic // and sets NumAtBarrier to 0
    NumGone++;
}

The WaitAtBarrier() Logic

Sample Simulation Output