Fish and Sharks
Hiromi Suenaga and Phil Merkey

Introduction

This program implements a synthetic predator-prey model. All decision with regard to the model and the design of this implementation were based on our desire to experiment with UPC. It is unlikely that this model has any correlation with actual predator-prey problems in the real world. In our model we have the fish swimming in a 2-dimensional ocean and being chased and eaten by sharks.

The first characteristic that we wanted to explore was the interaction between two different primary data structures. The collection of fish and sharks are stored as doubly linked lists of structures that record their position, velocity, age and hunger level. The movement of the fish and sharks is determined by “forces” that are calculated at each point of the 2-dimensional grid that represents the ocean.

The program is a single file that has a generic UPC program structure:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <upc_relaxed.h>

//defines and globals

//subroutines

//main
```

Global variable and Parameters

The ocean is a ORDER × ORDER grid. Note that the straightforward implementation of the Morton Z-ordering requires that the ORDER (order as in the order of a matrix) of the grid is a power of two. That is, the number of grid points is a power of four. To control the affinity of data within the grid, we associate subblocks of the grid with particular threads. In order to have the Z-ordering hierarchy match the subblocks structure so we must also set the value of BLOCK, the number of elements in each subblock, to be a power of four as well. In this way each THREAD has affinity to GRIDSIZE / (BLOCK*THREADS) subblocks. This is essentially a block cyclic decomposition of the grid.

```c
#define LOGORDER 8
#define ORDER (256)   // (1<<LOGORDER)
#define GRIDSIZE (65536) // (ORDER*ORDER)
#define BLOCK (1024)   // ((GRIDSIZE)/(BLKPERTHREAD*THREADS))
```

```c
#define LOGORDER 6
#define ORDER (64)    // (1<<LOGORDER)
#define GRIDSIZE (4096) // (ORDER*ORDER)
#define BLOCK (256)   // ((GRIDSIZE)/(BLKPERTHREAD*THREADS))
```
The manipulation of the dilated integers in the Morton Z-ordering requires the use of masks to strip off unwanted even and odd bits. If we shorten the masks to incorporate the grid size we automatically compute the wrap around for the toroidal grid. That is, if we use the right masks then we will have $0 = Jinc(Jxpod(ORDER-1))$. The binary expansion for $ORDER - 1$ has $LOGORDER$ ones, so the dilated representations should also have $LOGORDER$ ones.

\[Zorder defines and globals\]

\#define Imask 0x0000AAAA
\#define Jmask 0x00005555

The program stops if either the fish or the sharks become extinct or it runs for the maximum number of timesteps, NUMTIMESTEPS. At each timestep we compute the number of fish and sharks and center of mass of the fish. These values are stored in the shared variables FishPop, SharkPop and FishMassX and FishMassY.

\{defines and globals\}

\#define NUMTIMESTEPS 100

int RandomSeed = 1;

shared int FishMassX, FishMassY;
shared int FishPop, SharkPop;
All the information about the fish and sharks is stored in doubly linked lists of structures. This was chosen because the access to lists is always a loop over all the fish or sharks, i.e., we never have to search through the list and because we need to be able to add and remove members easily. Note that there is a list for each thread, and they are all independent of each other.

For each fish we record its age, velocity and its position in the ocean. Note that its position is recorded by pointing to the appropriate grid cell in the ocean. In addition to age, velocity and position we also record a shark’s hunger level. Sharks need to eat or they starve to death.

```plaintext
\langle defines and globals \rangle + \equiv

struct Fish {
    int age;
    int vx;
    int vy;
    shared [BLOCK] struct OceanGrid *position;
    struct Fish *next;
    struct Fish *before;
};

struct Shark {
    int age;
    int hungry;
    int vx;
    int vy;
    shared [BLOCK] struct OceanGrid *position;
    struct Shark *next;
    struct Shark *before;
};

struct Fish * FishList = NULL;
struct Shark * SharkList = NULL;

struct Fish * CreateFish(int, int, int);
struct Shark * CreateShark(int, int, int);
```
The tunable parameters that control the interaction between the fish and sharks are given below. The initial population of fish per thread is \texttt{InitFishPop} and no fish are born if the population is greater than \texttt{MaxFishPop}. Fish have a life expectancy of \texttt{FishLifeExp}. After that age they die with probability \texttt{FishDeathRate}/100. After fish reach maturity at age \texttt{FishMature} they reproduce (clone themselves) with probability \texttt{FishBirthRate}/100. The parameters \texttt{MinFishVelocity} and \texttt{MaxFishVelocity} limit how fast a fish can swim.

The corresponding parameters for the sharks are also listed here. In addition, the sharks must eat to survive. The parameter \texttt{FISHCALS} is how much a sharks hunger is reduced for each fish. Note that shark hunger is a positive measure, so sharks that are well fed have negative hunger.

\begin{verbatim}
\#define InitFishPop (GRIDSIZE/16)
\#define MaxFishPop  (GRIDSIZE/4)
\#define FishLifeExp (12)
\#define FishDeathRate (50)
\#define FishMature (5)
\#define FishBirthRate (60)
\#define MinFishVelocity (-2)
\#define MaxFishVelocity (2)
\#define FISHCALS (5)
\#define InitSharkPop (GRIDSIZE/128)
\#define MaxSharkPop (GRIDSIZE/16)
\#define SharkLifeExp (50)
\#define SharkDeathRate (40)
\#define SharkMature (5)
\#define SharkBirthRate (12)
\#define MinSharkVelocity (-4)
\#define MaxSharkVelocity (4)
\end{verbatim}
The Ocean is a 2-dimensional grid of structures. The 2-dimensional grid is laid out using the Morton Z-ordering. We store the dilated form of the coordinates so that we don’t need to recompute them. The components `nfish` and `nsharks` are the number of fish and sharks at that point. The `FishSmell` and `SharkSmell` is the weighted sum of the number of fish and sharks in a neighborhood of the grid point. At each time step we use the “smell” to compute a “force” on fish and sharks at that grid point. That force is then used by the fish and sharks to change their velocity.

The Ocean is stored in shared memory. The Z-ordering format naturally induces a checkerboard data distribution. In this case we are using a striped checkerboard layout because of the limitation on UPC block size in the original implementation on the T3E.

```c
<defines and globals>+≡
struct OceanGrid{
    int positionX;       // dilated form for X coordinate
    int positionY;       // dilated form for Y coordinate
    int nfish;
    int nsharks;
    int FishSmell;
    int SharkSmell;
    int FishForceX;
    int FishForceY;
    int SharkForceX;
    int SharkForceY;
};
```

```c
shared[ BLOCK ] struct OceanGrid Ocean[GRIDSIZE];
```
These routines are called whenever a fish or shark is born. They initialize the structure for a fish or shark and return a pointer to it. Note they return NULL if the malloc fails.

\[\text{subroutines}\equiv\]

\[
\text{struct Fish* CreateFish(int a, int I, int J)}
\{
    \text{struct Fish* fish;}

    \text{if( (fish = (\text{struct Fish*}) malloc (sizeof(\text{struct Fish}))) } \neq \text{ NULL)}
    \{
        \text{fish->age } = \text{ a;}
        \text{fish->vx } = \text{ 0;}
        \text{fish->vy } = \text{ 0;}
        \text{fish->position } = \&\text{Ocean[I+J];}
    \}
    \text{return fish;}
\}

\[
\text{struct Shark* CreateShark(int a, int I, int J)}
\{
    \text{struct Shark* shark;}

    \text{if( (shark = (\text{struct Shark*}) malloc (sizeof(\text{struct Shark}))) } \neq \text{ NULL)}
    \{
        \text{shark->age } = \text{ a;}
        \text{shark->vx } = \text{ 0;}
        \text{shark->vy } = \text{ 0;}
        \text{shark->hungry } = \text{ (int)random()\%100;}
        \text{shark->position } = \&\text{Ocean[I+J];}
    \}
    \text{return shark;}
\}
\]

\text{Main}

The main program simply initializes the lists and then runs the main time step loop.

\[\text{main}\equiv\]

\[
\text{int main(int args, char * argv[])}
\{
    \text{int timesteps=NUMTIMESTEPS;}
    \text{long sec, usec;}
    \text{struct timeval stamps [2];}

    \text{(check parameters)}
    \text{(initialize the list of fish and sharks and the ocean)}

    \text{(Main time step loop)}
\}
Initialize

Using the masking trick in the Morton Z-ordering to make the grid a torus works in a natural way if all the parameters are carefully chosen. The number of threads needs to be a power of 4. Currently we have access to machines with no more than 48 processors, so we only check to be sure that $\text{THREADS} = 1, 4, 16$. 

(check parameters)≡

```c
int stoprun = 0;

if ( !ispoweroffour( THREADS ) || THREADS > 16 ) {
    stoprun |= 1;
    fprintf(stderr,"check the number of THREADS\n");
}
if ( !ispoweroffour( BLOCK ) ) {
    stoprun |= 1;
    fprintf(stderr,"check the size of each block\n");
}
if( !ispoweroffour( GRIDSIZE ) ){
    stoprun |= 1;
    fprintf(stderr,"check the grid size\n");
}
if( stoprun )
    exit(1);
```
Each THREAD has its own list of fish and sharks stored in local memory. In this version of the program, there is no attempt to make these parallel lists mimic the behavior of the single linked list in the serial version of the program. This has a number of negative consequences: 1) we can’t duplicate the serial run, 2) if a list becomes empty there will be load balancing issues for the rest of the run, and 3) no attention is paid to the affinity of the fish and sharks. We plan to address issues 2 and 3 in future versions of the program.

The affinity issue is interesting because it is dynamic. The issue is that a fish that is stored in a list that is local to THREAD \( i \) can have a position (point to a grid point) that has affinity to THREAD \( j \) for arbitrary \( i \) and \( j \). One idea to improve performance is to add a sorting step to the main time loop. We would sort the lists so that all the fish and sharks that point to the ocean grid with affinity to thread \( i \) would be stored on the lists local to thread \( i \). Of course, as the fish and sharks move around the ocean their affinity changes. The question becomes a trade-off in overhead. Either we sort the list often and make few non-affinity memory references or we allow the affinity to change and avoid the sorting overhead.

There are also plans to move the linked lists into shared memory. This would help with problems 1 and 2, but we would still have to handle issues involving affinity.

After initializing the lists of fish and sharks we initialize the grid by computing and storing the dilated format for the coordinates of the grid points.

\[
\text{(initialize the list of fish and sharks and the ocean)} = \\
\begin{align*}
\text{\{ int } i, K; \\
\text{ struct Fish } *fish; \\
\text{ struct Shark } *shark; \\
\text{srandom(1+MYTHREAD); \\

\text{ for(i=0; i<InitFishPop; i++)} \\
\text{ \{ \\
\text{ fish = CreateFish( \\
\text{ (int)random()}%FishLifeExp, \\
\text{ Ixpnd((int)random()}%ORDER), \\
\text{ Jxpnd((int)random()}%ORDER)); \\
\text{ if(FishList != NULL){ \\
\text{ FishList->before = fish; \\
\text{ } } \\
\text{ fish->next = FishList; \\
\text{ fish->before = NULL; \\
\text{ FishList = fish; \\
\text{ } } } \}
\text{ \\
\text{ for(i=0; i<InitSharkPop; i++)} \\
\text{ \{ \\
\text{ shark = CreateShark( (int)random()}%SharkLifeExp, \\
\text{ Ixpnd((int)random()}%ORDER), \\
\text{ Jxpnd((int)random()}%ORDER)); \\
\text{ if(SharkList != NULL) \\
\text{ SharkList->before = shark; \\
\text{ shark->next = SharkList; \\
\text{ shark->before = NULL; \\
\text{ SharkList = shark; \\
\text{ } } } \}
\end{align*}
\]
Main time loop
The main time step loop is the heart of the program. At each time step, we use the position of the fish and sharks to create a force field on the grid. Then we use the force field to push the fish and sharks to their new positions.

\[ \text{(Main time step loop)} \equiv \]
\[ \text{do} \]
\[ \{ \text{(Compute force vectors at each grid point)} \]
\[ \text{(Update the fish list)} \]
\[ \text{(Update the shark list)} \]
\[ \text{(Print Output)} \]
\[ \} \text{ while( timesteps-->0 );} \]

Working with the grid based data
The computation in this section of the code is focused on regular grid based data. Placing the fish and the sharks on the grid amounts to random writes into the shared space. The rest of the calculations are completely regular and parallelizes easily.

\[ \text{(Compute force vectors at each grid point)} \equiv \]
\[ \text{(Place fish and sharks on the grid and find center of mass)} \]
\[ \text{(Compute Fish and Shark smell)} \]
\[ \text{(Use smell to determine the force vectors)} \]
In this section we rebuild the grid and count the number of fish and number of sharks at each grid point.

The first thing to do is zero out the counts from the previous time step.

\(\text{Place fish and sharks on the grid and find center of mass}\) ≡

```c
{ int i, j, k;
  int myfishpop, mysharkpop;
  int myfishmassX, myfishmassY;
  struct Fish *fish;
  struct Shark *shark;

  myfishpop = 0;
  mysharkpop = 0;
  myfishmassX = 0;
  myfishmassY = 0;

  if(MYTHREAD == 0){
    FishPop = 0;
    SharkPop = 0;
    FishMassX = 0;
    FishMassY = 0;
  }

  upc_forall(k=0; k<GRIDSIZE; k++; &Ocean[k]){ // Clear the grid
    Ocean[k].nfish = 0;
    Ocean[k].nsharks = 0;
    Ocean[k].FishSmell = 0;
    Ocean[k].SharkSmell = 0;
  }

  upc_barrier;
```
The second step is to record the number of fish and sharks at each grid point. Recall that in this implementation the list of fish and sharks are stored in local memory for each thread. So in this case, updating the grid is equivalent to each thread incrementing random locations in shared memory. The question is how to handle the potential write conflicts given that we have a small number of threads writing into a large array.

One approach was to associate a lock with each grid point. Then one would acquire that specific lock before updating the value. This approach has no false sharing of locks, but is also requires as many locks as grid points. The other extreme is to have one lock for the whole grid. This of course would serialize the process.

The approach taken in this code is to simply ignore the locks. In this case we lose some of the updates. The number we lose is in proportion to conflicts. Preliminary studies showed that we were only losing one or two percent of the fish. Since we only need approximate values to compute the forces in the next section, this approach seems reasonable.

Note that none of this discussion translates into the message passing model. In a message passing model one would probably sort the fish by affinity and then send bulk messages to the appropriate processors to do the updates. Once you have paid for the overhead of sorting and collection bulk messages, none of the fine grained strategies that we can exploit make sense in that model.

\[\text{Place fish and sharks on the grid and find center of mass}\]

\[
\begin{align*}
\text{for(fish=FishList; fish!=NULL; fish=fish->next) } \{ \\
\quad \text{myfishpop++;} \\
\quad \text{fish->position->nfish +=1; } \quad \text{\textit{//WARNING: NO LOCKS}} \\
\} \\
\text{for(shark=SharkList; shark!=NULL; shark=shark->next) } \{ \\
\quad \text{mysharkpop++;} \\
\quad \text{shark->position->nsharks +=1; } \quad \text{\textit{//WARNING: NO LOCKS}} \\
\}
\end{align*}
\]
Now using the values in the grid and the exact population counts, we compute
the center of mass for the fish. The center of mass of the fish is used to help the
sharks in case there is nothing near by to attack.

(\textit{defines and globals}) \equiv
\begin{verbatim}
upc_lock_t centroidlock;
\end{verbatim}

(\textit{Place fish and sharks on the grid and find center of mass}) \equiv
\begin{verbatim}
upc_barrier;
upc_forall (k=0; k<GRIDSIZE; k++; &Ocean[k]){ 
  myfishmassX += ixtrct(k)*Ocean[k].nfish;
  myfishmassY += jxtrct(k)*Ocean[k].nfish;
}
upc_lock(&centroidlock);
FishPop += myfishpop;
SharkPop += mysharkpop;
FishMassX += myfishmassX;
FishMassY += myfishmassY;
upc_unlock(&centroidlock);
upc_barrier;
if( (FishPop == 0) || (SharkPop == 0 ) ){
  exit(2);
}
if(MYTHREAD == 0){
  FishMassX = FishMassX/FishPop;
  FishMassY = FishMassY/FishPop;
}
\end{verbatim}

The Fish and Shark smell is a way to do something like an inverse square law. Each fish or shark contributes a smell of 3 to its grid point, a smell of 2 to the grid points one away and 1 to the grid points 2 away. The forces are computed in the next step based on the smell.

Instead of writing complicated loops to specify the neighborhood of a grid point, we simply wrote straight line code for all 25 grid points in the neighborhood. It is a little verbose, but it exposes a lot of ILP.

\[
\text{(Compute Fish and Shark smell)} \equiv \\
\text{\{ int I,J,K;}
\text{ int Im2,Ip2,Im1,Ip1,Jm2,Jp2,Jm1,Jp1,sharknum, fishnum;}
\text{ upc_forall(K=0; K<GRIDSIZE; K++; &Ocean[K]){}
\text{ fishnum = Ocean[K].nfish;}
\text{ sharknum = Ocean[K].nsharks;}
\text{ I = K&Imask;}
\text{ J = K&Jmask;}
\text{ Im2 = (I - 8) & Imask;}
\text{ Im1 = (I - 2) & Imask;}
\text{ Ip1 = ((I&Jmask)+2) & Imask;}
\text{ Ip2 = ((I&Jmask)+8) & Imask;}
\text{ Jm2 = (J - 4) & Jmask;}
\text{ Jm1 = (J - 1) & Jmask;}
\text{ Jp1 = ((J&Imask)+1) & Jmask;}
\text{ Jp2 = ((J&Imask)+4) & Jmask;}
\text{ Ocean[ I+ J].FishSmell += 3*fishnum;}
\text{ Ocean[Im1+Jm1].FishSmell += 2*fishnum;}
\text{ Ocean[Im1+ J].FishSmell += 2*fishnum;}
\text{ Ocean[Im1+Jp1].FishSmell += 2*fishnum;}
\text{ Ocean[ I+Jm1].FishSmell += 2*fishnum;}
\text{ Ocean[ I+Jp1].FishSmell += 2*fishnum;}
\text{ Ocean[Jp1+Jm1].FishSmell += 2*fishnum;}
\text{ Ocean[Jp1+ J].FishSmell += 2*fishnum;}
\text{ Ocean[Jp1+Jp1].FishSmell += 2*fishnum;}
\text{ Ocean[Jp1+Jm2].FishSmell += fishnum;}
\text{ Ocean[Jm2+Jm2].FishSmell += fishnum;}
\text{ Ocean[Jm2+ J].FishSmell += fishnum;}
\text{ Ocean[Jm2+Jp1].FishSmell += fishnum;}
\text{ Ocean[Jm2+Jp2].FishSmell += fishnum;}
\text{ Ocean[ Jm2].FishSmell += fishnum;}
\text{ Ocean[Ip1+Jm2].FishSmell += fishnum;}
\text{ Ocean[Im1+Jp2].FishSmell += fishnum;}
\text{ Ocean[ I+Jp2].FishSmell += fishnum;}
\text{ Ocean[Ip1+Jp2].FishSmell += fishnum;}
\text{ Ocean[Ip2+Jm2].FishSmell += fishnum;}
\text{ Ocean[Ip2+Jm1].FishSmell += fishnum;}
\text{ Ocean[Ip2+ J].FishSmell += fishnum;}
\text{ Ocean[Ip2+Jp1].FishSmell += fishnum;}
\text{ Ocean[Ip2+Jp2].FishSmell += fishnum;}
\text{ Ocean[ J].SharkSmell += 3*sharknum;}
\text{ Ocean[Im1+Jm1].SharkSmell += 2*sharknum;}
\text{ Ocean[Im1+ J].SharkSmell += 2*sharknum;}
\}}
\]
Ocean[Im1+Jp1].SharkSmell += 2*sharknum;
Ocean[ I+Jm1].SharkSmell += 2*sharknum;
Ocean[ I+Jp1].SharkSmell += 2*sharknum;
Ocean[Ip1+Jm1].SharkSmell += 2*sharknum;
Ocean[Ip1+ J].SharkSmell += 2*sharknum;
Ocean[Ip1+Jp1].SharkSmell += 2*sharknum;
Ocean[Im2+Jm1].SharkSmell += sharknum;
Ocean[Im2+Jm1].SharkSmell += sharknum;
Ocean[Im2+ J].SharkSmell += sharknum;
Ocean[Im2+Jp1].SharkSmell += sharknum;
Ocean[Im2+Jp2].SharkSmell += sharknum;
Ocean[Im1+Jm2].SharkSmell += sharknum;
Ocean[ I+Jm2].SharkSmell += sharknum;
Ocean[Ip1+Jm2].SharkSmell += sharknum;
Ocean[ I+Jp2].SharkSmell += sharknum;
Ocean[ Ip1+Jp2].SharkSmell += sharknum;
Ocean[Ip2+Jm2].SharkSmell += sharknum;
Ocean[Ip2+Jm1].SharkSmell += sharknum;
Ocean[Ip2+ J].SharkSmell += sharknum;
Ocean[Ip2+Jp1].SharkSmell += sharknum;
Ocean[Ip2+Jp2].SharkSmell += sharknum;
}
upc_barrier;
}
To compute the force vectors we look at the $5 \times 5$ neighborhood of each grid point to find (the first maximal) fish and shark smell. Fish get a force away from the maximal sharks smell or a random force if no sharks are present. Sharks get a force toward the maximal fish smell or toward the center of mass of the fish if no fish smell is present.

\[ \text{Use smell to determine the force vectors} \equiv \]

\begin{verbatim}
upc_forall(K=0; K<GRIDSIZE; K++; &Ocean[K]){  
    I = K&Imask;  
    J = K&Jmask;  
    MaxfishSmell=0;  
    MaxsharkSmell=0;  
    MaxfishSmellPosX = 0;  
    MaxfishSmellPosY = 0;  
    MaxsharkSmellPosX = 0;  
    MaxsharkSmellPosY = 0;  

    Ineigh[0] = (I - 8) & Imask;  
    Ineigh[1] = (I - 2) & Imask;  
    Ineigh[2] = I;  
    Ineigh[3] = ((I|Jmask)+2) & Imask;  
    Ineigh[4] = ((I|Jmask)+8) & Imask;  

    Jneigh[0] = (J - 4) & Jmask;  
    Jneigh[1] = (J - 1) & Jmask;  

    for(di=0; di<5; di++){  
        for(dj=0; dj<5; dj++){  

            if(Ocean[Ineigh[di] + Jneigh[dj] ].FishSmell > MaxfishSmell) {  
                MaxfishSmellPosX = di-2;  
                MaxfishSmellPosY = dj-2;  
            }  

            if(Ocean[Ineigh[di] + Jneigh[dj] ].SharkSmell > MaxsharkSmell) {  
                MaxsharkSmellPosX = di-2;  
                MaxsharkSmellPosY = dj-2;  
            }  
        }  
    }  

    if( MaxsharkSmell == 0 ){
        if ( MaxfishSmellPosX > 0 )

    }

}
\end{verbatim}
Ocean[K].FishForceX = 1; 
else if (MaxfishSmellPosX < 0)
  Ocean[K].FishForceX = -1;
else
  Ocean[K].FishForceX = ((int)random()%(3)-1;
if (MaxfishSmellPosY > 0)
  Ocean[K].FishForceY = 1;
else if (MaxfishSmellPosY < 0)
  Ocean[K].FishForceY = -1;
else
  Ocean[K].FishForceY = ((int)random()%(3)-1;
}
else{
  if (MaxsharkSmellPosX > 0)
    Ocean[K].FishForceX = -1;
  else if (MaxsharkSmellPosX < 0)
    Ocean[K].FishForceX = 1;
  else
    Ocean[K].FishForceX = 0;
  if (MaxsharkSmellPosY > 0)
    Ocean[K].FishForceY = -1;
  else if (MaxsharkSmellPosY < 0)
    Ocean[K].FishForceY = 1;
  else
    Ocean[K].FishForceY = 0;
}

if (MaxfishSmell == 0) { // move to center of mass
  if (FishMassX - ixtrct(I) == 0)
    Ocean[K].SharkForceX = 0;
  else
    Ocean[K].SharkForceX = ((FishMassX - ixtrct(I)) * (abs(2*(FishMassX-ixtrct(I)))-ORDER) > 0) ? -1 : 1;
  if (FishMassY - jxtrct(J) == 0)
    Ocean[K].SharkForceY = 0;
  else
    Ocean[K].SharkForceY = ((FishMassY - jxtrct(J)) * (abs(2*(FishMassY-jxtrct(J)))-ORDER) > 0) ? -1 : 1;
}
else{
  Ocean[K].SharkForceX = MaxfishSmellPosX;
  Ocean[K].SharkForceY = MaxfishSmellPosY;
}

upc_barrier;
Update the fish and shark lists

In this section of the time step loop we update the lists of fish and sharks maintained in each thread. Most of the work in this section is serial code within each thread to manipulate the lists along with a series of random reads into the shared values held in the grid. The number of fish and sharks at a grid point and forces on the fish and sharks are needed in this process.

Fish are eaten if they share a grid point with a shark. Sharks satisfy their hunger by eating the fish. If the fish and sharks survive, the forces are used to determine their new velocities and the new velocities determine their new positions.

\textit{Update the fish list} ≡
{ struct Fish *fish, *babyfish;

    for(fish=FishList; fish!=NULL; fish=fish->next)
    {
      if (the fish dies)
        (remove fish from list)
      else
        (move the fish and spawn new fish)
    }
}

\textit{Kill the fish if it is too old or there is a shark at this grid point.}
(\textit{the fish dies}) ≡
( ( fish->position->nsharks > 0 ) ||
 ( (fish->age > FishLifeExp) && ( (random()%100) < FishDeathRate ) )
)

\textit{Standard serial link list manipulation.}
\textit{(remove fish from list)} ≡
{ if(fish->before==NULL) { /* the only fish in the list */
      FishList = NULL;
    } else {
      fish->next->before = NULL;
      FishList = fish->next;
    }
    } else {
      if(fish->next == NULL) /* at least one fish before him but none after */
        fish->before->next = NULL;
      else{
        fish->before->next = fish->next;
        fish->next->before = fish->before;
      }
    }
    free(fish);
}
If the fish survives then we use the force on the fish to change its velocity (up to its maximum velocity). Recall that the position of the fish is maintained by the pointer into the grid. So, moving the fish means updating this pointer.

Also in this section we spawn new fish. If a fish has reached maturity, Fish-Mature, it spawns a new fish with probability FishBirthRate. The new fish is randomly placed at a grid point near the parent fish.

(move the fish and spawn new fish) ≡
{ int I,J,newI,newJ;
  int iforce,jforce;
  struct Fish *babyfish;

  I = fish->position->positionX;
  J = fish->position->positionY;

  fish->vx += fish->position->FishForceX;
  if( fish->vx > MaxFishVelocity )
    fish->vx = MaxFishVelocity;
  if( fish->vx < MinFishVelocity )
    fish->vx = MinFishVelocity;
  fish->vy += fish->position->FishForceY;
  if( fish->vy > MaxFishVelocity )
    fish->vy = MaxFishVelocity;
  if( fish->vy < MinFishVelocity )
    fish->vy = MinFishVelocity;

  if( fish->vx == -2 )
    I = (I - 8) & Imask;
  else if( fish->vx == -1 )
    I = (I - 2) & Imask;
  else if( fish->vx == 1 )
    I = ((I|Jmask)+2) & Imask;
  else if (fish->vx == 2 )
    I = ((I|Jmask)+8) & Imask;
  else
    ;
  if( fish->vy == -2 )
    J = (J - 4) & Jmask;
  else if( fish->vy == -1 )
    J = (J - 1) & Jmask;
  else if( fish->vy == 1 )
    J = ((J|Imask)+1) & Jmask;
  else if (fish->vy == 2 )
    J = ((J|Imask)+4) & Jmask;
  else
    ;
  fish->position = &Ocean[I+J];
(move the fish and spawn new fish) +≡
if ( fish->age > FishMature &&
( FishPop < MaxFishPop ) && ((random())%100) <= FishBirthRate ) {
    switch( random()%3 ) {
        case 1:
            newI = (I-2) & Imask; break;
        case 2:
            newI = (I|Jmask+2) & Imask; break;
        default:
            newI = I;
    }
    switch( random()%3 ) {
        case 1:
            newJ = (J-1) & Jmask; break;
        case 2:
            newJ = (J|Imask+1) & Jmask; break;
        default:
            newJ = J;
    }
    babyfish = CreateFish(0, newI, newJ);
    babyfish->next = FishList;
    babyfish->before = NULL;
    FishList->before = babyfish;
    FishList = babyfish;
    fish->age +=1;
}

Updating the shark list is similar to updating the fish list with the additional constraint that sharks die if they get too hungry.

(Update the shark list) ≡
{ struct Shark *shark;

    for(shark=SharkList; shark!=NULL; shark=shark->next) {
        if((shark dies))
            (remove shark from list)
        else {
            (feed and move shark and make new shark)
        }
    }

    (shark dies) ≡
    ( shark->hungry >= 100 ||
        shark->age > SharkLifeExp ||
        ( (SharkPop > MaxSharkPop) && (random())%100 < SharkDeathRate) )
}
(remove shark from list)\[\{ \\
  if(shark->before==NULL) \\
    \{ \\
      if(shark->next == NULL) /* that was the only shark in the list */ \\
        SharkList = NULL; \\
      else{ /* there is at least one shark after him */ \\
        shark->next->before = NULL; \\
        SharkList = shark->next; \\
      } \\
    \} \\
  else{ \\
    if(shark->next == NULL){ /* at least one shark before him but none after */ \\
      shark->before->next = NULL; \\
    } \\
    else{ \\
      shark->before->next = shark->next; \\
      shark->next->before = shark->before; \\
    } \\
  } \\
  free(shark); \\
\}
{(feed and move shark and make new shark)≡

{ int I,J,newI,newJ;
  int Idil[5]={0,2,8,10,32};
  int Jdil[5]={0,1,4,5,16};

  struct Shark *babyshark;

  I = shark->position->positionX;
  J = shark->position->positionY;

  shark->vx += shark->position->SharkForceX;
  if( shark->vx > MaxSharkVelocity )
    shark->vx = MaxSharkVelocity;
  if( shark->vx < MinSharkVelocity )
    shark->vx = MinSharkVelocity;
  shark->vy += shark->position->SharkForceY;
  if( shark->vy > MaxSharkVelocity )
    shark->vy = MaxSharkVelocity;
  if( shark->vy < MinSharkVelocity )
    shark->vy = MinSharkVelocity;

  if( shark->vx < 0 )
    I = ( I - Idil[(-1)*shark->vx] ) & Imask;
  if( shark->vx > 0 )
    I = ((I|Jmask) + Idil[shark->vx] ) & Imask;
  if( shark->vy < 0 )
    J = (J - Jdil[(-1)*shark->vy]) & Jmask;
  if( shark->vy > 0 )
    J = ((J|Imask) + Jdil[shark->vy]) & Jmask;

  if( shark->position->nfish>0 )
    shark->hungry -= shark->position->nfish * FISHCALS;

  shark->position = &Ocean[ I+J ];}
(feed and move shark and make new shark) +≡
   if( (shark->age > SharkMature) && (shark->hungry < 0) &&
       (random()%100)<= SharkBirthRate ) {
       switch( random()%3 ) {
           case 1:
               newI = (I-2) & Imask; break;
           case 2:
               newI = (I|Jmask+2) & Imask; break;
           default:
               newI = I;
       }
       switch( random()%3 ) {
           case 1:
               newJ = (J-1) & Jmask; break;
           case 2:
               newJ = (J|Imask+1) & Jmask; break;
           default:
               newJ = J;
       }
       babyshark = CreateShark(0,newI,newJ);
       babyshark->next = SharkList;
       babyshark->before = NULL;
       SharkList->before = babyshark;
       SharkList = babyshark;
   }
   shark->hungry += 1;
   shark->age += 1;
}

Printing Output
We have several output formats that could be used. We choose between them by
changing the name of the macro to read Print Output.
   The simple output is just printing the center of mass and the population counts
for the fish and the sharks.
   (Print Output Center of Mass) ≡
   if( MYTHREAD == 0 ){
       printf("%d %d %d %d\n",FishMassX, FishMassY, FishPop, SharkPop);
   }
upc_barrier;
There is a java program that reads a file containing the list of all the fish and sharks and then plots them on a grid. To feed that program we use the following output format.

(Print Output from lists) ≡
{
    struct Fish *fish;
    struct Shark *shark;
    if (MYTHREAD == 0 ){
        printf("FP %i\nSP %i\n", FishPop, SharkPop);
    }
    upc_barrier;
    for(fish=FishList; fish!=NULL; fish=fish->next) {
        printf("F %d %d\n", ixtrct(fish->position->positionX),
               jxtrct(fish->position->positionY));
    }
    upc_barrier;
    for(shark=SharkList; shark!=NULL; shark=shark->next) {
        printf("S %d %d\n", ixtrct(shark->position->positionX),
               jxtrct(shark->position->positionY));
    }
    upc_barrier;
}

There is a post processor that reads a trace of the execution and displays a movie of what happened.

(Print Output) ≡
upc_barrier;
if(MYTHREAD == 0 )
{
    int i,j;
    for(i=0; i<ORDER; i++){
        for(j=0; j<ORDER; j++){
            if( (Ocean[Zord(i,j)].nfish != 0) || (Ocean[Zord(i,j)].nsharks != 0) )
                printf("%c%c%c", i&0xFF, j&0xFF ,
                        Ocean[Zord(i,j)].nfish & 0x7F, Ocean[Zord(i,j)].nsharks & 0x7F);
        }
    }
    printf("%c%c%c",0x0,0x0,0x0,0xFF);
}
upc_barrier;
Z-order Support Macros and Routines

The following macros and subroutines support the use of Morton Z-ordering for the ocean grid. The macros \texttt{Imask} and \texttt{Jmask} were defined in the section that sets the size of the grid and the blocking factors. The rest of the macros are standard support routines for the Z-ordering.

\begin{verbatim}
#define Zord(i,j) ( (expand((i))<<1) | expand((j)) )
#define ixtrct(K) (extract( ((K))>>1 ))
#define jxtrct(K) (extract( ((K)) )
#define Ixpnd(k) (expand((k))<<1)
#define Jxpnd(k) (expand(k))
#define Iinc(I) ( (((I)|Jmask)+2) & Imask )
#define Jinc(J) ( (((J)|Imask)+1) & Jmask )
\end{verbatim}

These subroutines convert between integer in the standard and the (even) di-
dilated format.

\begin{verbatim}
expand( int t )
{
    int u,v,w,x;
    u = ((t&0x0000FF00)<<8) | (t & 0x000000FF);
    v = ((u&0x00F000F0)<<4) | (u & 0x000F000F);
    w = ((v&0x0C0C0C0C)<<2) | (v & 0x03030303);
    x = ((w&0x22222222)<<1) | (w & 0x11111111);
    return(x);
}
extract( int t )
{
    unsigned u,v,w,y,x;
    u = (t&0x55555555);
    v = ((u&0x44444444)>>1) | (u & 0x11111111);
    w = ((v&0x30303030)>>2) | (v & 0x03030303);
    x = ((w&0x0F000F00)>>4) | (w & 0x000F000F);
    y = ((x&0x00FF0000)>>8) | (x & 0x000000FF);
    return((int)y);
}
\end{verbatim}
Support and Debugging Routines

The first few functions are simple print routines used in the debugging process.

```c
int ispoweroffour(int n)
{
    // only check to 4\(^{-15}\)th, 
    // cause we want to avoid word size problems
    int i, p;
    p = 1;
    for( i=0; i<15; i++){
        if( p == n)
            return(1);
        p *= 4;
    }
    return(0);
}

printoceansmell()
{
    int i, j;
    printf("\nsmell\n");
    for(i=0; i<ORDER; i++){
        for(j=0; j<ORDER; j++){
            printf("%2d ", Ocean[Zord(i,j)].FishSmell);
        }
        printf(" ");
    }
    printf("
");
    for(j=0; j<ORDER; j++){
        printf("%2d ", Ocean[Zord(i,j)].SharkSmell);
    }
    printf("\n");
    printf("\n-----------\n");
```
⟨subroutines⟩+≡
printoceanforce()
{
    int i,j;
    printf("\nforce\n");
    for(i=0; i<ORDER; i++){
        for(j=0; j<ORDER; j++){
            printf("%2d ", Ocean[Zord(i,j)].FishForceX);
        }
        printf(" ");
        for(j=0; j<ORDER; j++){
            printf("%2d ", Ocean[Zord(i,j)].FishForceY);
        }
        printf("n");
    }
    printf("n");
    for(i=0; i<ORDER; i++){
        for(j=0; j<ORDER; j++){
            printf("%2d ", Ocean[Zord(i,j)].SharkForceX);
        }
        printf(" ");
        for(j=0; j<ORDER; j++){
            printf("%2d ", Ocean[Zord(i,j)].SharkForceY);
        }
        printf("n");
    }
    printf("n---------n");
}
⟨subroutines⟩+≡
printocean()
{
    int i, j;
    printf("nnfish and sharks  %d %d %d %d\n", 
        FishMassX, FishMassY, FishPop, SharkPop);
    for(i=0; i<ORDER; i++){
        for(j=0; j<ORDER; j++)
            printf("%2d ", Ocean[Zord(i,j)].nfish);
        printf(" ");
        for(j=0; j<ORDER; j++)
            printf("%2d ", Ocean[Zord(i,j)].nsharks);
        printf("\n");
    }
    printf("\n-----------\n");
    /*
for(i=0; i<ORDER; i++){
    for(j=0; j<ORDER; j++){
        printf("%2d %2d: %4d %4d %4d %4d %4d %4d %4d %4d
", 
            i, j,  
            Ocean[i][j].nfish, 
            Ocean[i][j].FishSmell, 
            Ocean[i][j].FishForceX, 
            Ocean[i][j].FishForceY, 
            Ocean[i][j].nsharks, 
            Ocean[i][j].SharkSmell, 
            Ocean[i][j].SharkForceX, 
            Ocean[i][j].SharkForceY );
    } }
*/
}
}