Universität Hamburg

Fachbereich Informatik

Ausarbeitung

Fish & Shark

im Praktikum parallele Programmierung

Thema:	Erstellung einer parallelen Simulation und – die Auswertung deren Performance
eingereicht von:	Sebastian Rothe <0rothe@informatik.uni-hamburg.de> Alexander Lambertz <0lambert@informatik.uni-hamburg.de>
eingereicht am:	29. Oktober 2012
Betreuer:	Julian Martin Kunkel Nathanael Hübbe

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieses Praktikums war unsere Aufgabe ein paralleles Programm, unter Benutzung der Programmiersprache C und dem Framework MPI, zu entwickeln. Wir haben uns dazu entschieden eine Simulation in Anlehnung an die Theorie von Wator zu programmieren. Es wird ein 2-dimensionales Spielfeld simuliert, auf dem sich Fische, Haie und Plankton befinden. Fische entstehen jeweils zufällig in bestimmten Zeitintervallen. Haie fressen Fische und vermehren sich dadurch. Außerdem haben Haie eine Lebensdauer, die bei der Entstehung aus einem vordefinierten Bereich ermittelt wird. Das Plankton vergrößert sich in bestimmten Zeitintervallen.

Unsere Simulation ist in zwei getrennte Programme aufgeteilt. Ein Programm führt die eigentliche Simulation durch und das zweite Programm führt im Anschluss die Umwandlung der Daten aus der Siumulation in Grafiken aus. Durch diese Aufteilung ist es uns möglich uns nur auf die Parallelisierung der Simulation zu konzentrieren und die Visualisierung in sequentiellem Code zu belassen. Außerdem ist dieses Szenario an der Realität orientiert, in der die Simulation einen höheren Stellenwert hat als die Visualisierung.

Die in der Simulation generierten Daten werden also in regelmäßigen Abschnitten in Txt-Dateien exportiert. Aus diesen Txt-Dateien erstellt dann die Visualisierung BMP-Dateien. Wir haben die Bilder im Anschluss zu einen Video zusammengeführt. Einige dieser Videos sind auf folgendem Youtube-Channel zu finden: `http://www.youtube.com/playlist?list=PLE3224435A7ED2859`

2 Design/Implementierung

2.1 Datenmodell

Für unsere Simulation nutzen wir ein 2-dimensionales Spielfeld. Dieses wird in einem 3-dimensionalen Array abgebildet. Die erste und zweite Dimension des Arrays stellen die y- bzw. x-Achse des Simulationsfeldes dar. In der dritten Dimension verfügt der Array je über 3 Werte:

1. Art des Elements in diesem Bereich
2. Todeszeitpunkt für Fische und Haie/Vermehrungszeitpunkt für Plankton
3. Flag zum speichern, ob der Bereich im aktuellen Zeitpunkt bereits bearbeitet wurde

In der Simulation wird der Array wie folgt initialisiert:

```
655     int ***aField;
656
657     aField = malloc(sizeof(int *) * iFieldSizeY);
658
659     if (aField == NULL)
660     {
661         printf("Memory Issue");
662         exit(1);
663     }
664
665     for (int y=0;y<iFieldSizeY;y++)
666     {
667         aField[y] = malloc (sizeof(int *) * iFieldSizeX);
668         // Set all values to 0
669         for (int x=0;x<iFieldSizeX;x++)
670         {
671             aField[y][x] = malloc (sizeof(int) * 3);
672             // Initialize 3 int-Values for each Field-Position
673             // Position 0 := Value of Sector
674             // Position 1 := DieTime
675             // Position 2 := Touched this round
676             aField[y][x][0] = 0;
677             aField[y][x][1] = -1;
678             aField[y][x][2] = 0;
679         }
680     }
```

main.c

Zuerst wird Speicherplatz für die Integer-Pointer des Arrays und für die 3 Integer in der dritten Dimension alloziert. Anschließend werden die Integer initialisiert. aField[y][x][1] wird mit -1 initialisiert, da sonst bereits im ersten Bild leere Bilder gelöscht würden. Dies wirkt sich negativ auf die Laufzeit des Programms aus.

Außer dem Spielfeld-Array brauchten wir für die Umsetzung unserer MPI-Kommunikation vier weitere Arrays, die die Randbereiche der Nachbarprozesse abbilden.

```
689     int *aBorderTop;
690     int *aBorderRight;
691     int *aBorderLeft;
692     int *aBorderBottom;
693
694     // +2 because Boder area is 2 Steps bigger then the normal area!!
695     // TODO: Redundance; not needed but not critical
696     aBorderTop = malloc(sizeof(int) * (iFieldSizeX));
697     aBorderBottom = malloc(sizeof(int) * (iFieldSizeX));
698
699     if (aBorderTop == NULL || aBorderBottom == NULL)
700     {
701         printf("Memory Issue");
702         // Free Memory
703         freeMemory(aField, aBorderTop, aBorderRight, aBorderBottom,
                aBorderLeft);
704         exit(1);
705     }
706
707     for (int x=0;x<(iFieldSizeX);x++)
708     {
709         aBorderTop[x] = 0;
710         aBorderBottom[x] = 0;
711     }
712
713     aBorderLeft = malloc(sizeof(int) * (iFieldSizeY));
714     aBorderRight = malloc(sizeof(int) * (iFieldSizeY));
715
716     if (aBorderLeft == NULL || aBorderRight == NULL)
717     {
718         printf("Memory Issue");
719         // Free Memory
720         freeMemory(aField, aBorderTop, aBorderRight, aBorderBottom,
                aBorderLeft);
721         exit(1);
722     }
723
724     for (int y=0;y<(iFieldSizeY);y++)
725     {
726         aBorderLeft[y] = 0;
727         aBorderRight[y] = 0;
728     }
```

main.c

In der folgenden Grafik wird dargestellt wie die Arrays angeordnet sind. In gelb sieht man die Border-Arrays, die die Randbereiche der Nachbarprozesse speichern und in blau ist der Simulationsarray dargestellt.

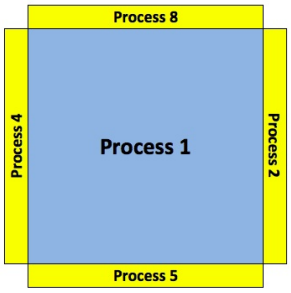

Abbildung 2.1: Nachbarschaft eines Prozesses (Beispiel 8 Prozesse)

2.2 Bearbeitung des Spielfeldes

Wir bearbeiten das Simulationsfeld in je drei ineinander geschachtelten Schleifen.

```
902 for (int iIntervalCount=0; iIntervalCount<iIntervalCountMax; iIntervalCount
    ++) {
```

main.c

Die erste Schleife steuert die Anzahl der Runden. Die Variable iIntervalCountMax stellt die maximale Anzahl der Runden dar und ist als Konstante zu Beginn des Programms definiert.

```
969         for (int i=0; (i<iFieldSizeY) && (iAmountAll>=iAmountProcessed); i
                ++) {
970             for (int j=0; (j<iFieldSizeX) && (iAmountAll>=iAmountProcessed)
                    ; j++) {
```

main.c

Die weiteren zwei Schleifen itterieren das Simulationsfeld. Außerdem verfügen die Schleifen über eine Bedingung, die überprüft, ob sich noch nicht bearbeitete Elemente auf dem Spielfeld befinden.

```
1111                // (aField[i][j][2]==0) := This means the field is touched
                       for the first time this round
1112                if ( (aField[i][j][0] > 0) && (aField[i][j][2]==0) )
```

main.c

In jedem Bereich wird überprüft, dass der Bereich nicht leer ist und, dass der Bereich in dieser Runde noch nicht bearbeitet wurde. Letztere Überprüfung ist wichtig, da in dem Fall, dass ein Fisch nach rechts bewegt wird, dieser im nächsten Bereich erneut bearbeitet würde.

Immer wenn im folgenden eine neue Position ermittelt wird, verwenden wir folgende Bedingung ($iRnds < iRndToZero$). Diese Bedingung haben wir zur Optimierung der Laufzeit eingesetzt. Zu Beginn des Programms kann die Konstante iRndToZero gesetzt werden, die die maximalen Versuche definiert, die unternommen werden um eine neue Position zu finden.
TODO

1. iRnds < iRndToZero)
2. (x!=0) || (y!=0)
3. checkIfFree(aField, aBorderTop, aBorderRight, aBorderBottom, aBorderLeft, i+y, j+x)==0

2.2.1 Haie

Im Folgenden gehen wir auf die Bearbeitung der Haie im Simulationsfeld ein. Die Lücken in den Codeabschnitten dienen dem besseren Verständnis. Außerdem wurden die Teile der MPI-Kommunikation entfernt, da diese im Abschnitt 2.3 „MPI Kommunikation“, S. 15 erläutert werden.
Bei der Bearbeitung eines Hais gilt es zwei Szenarien zu unterscheiden:

1. Es ist ein Fisch in den Nachbarfeldern verfügbar
2. Es ist kein Fisch in einem Nachbarfeld verfügbar

Diese Überprüfung führt die Funktion checkIfFishAround durch. Die Ausgabewerte dieser Funktion sind 1, wenn ein Fisch gefunden wird und 0, falls kein Fisch verfügbar ist. Falls ein Fisch gefunden wurde, werden außerdem die Werte x und y auf die Position des Fisches gesetzt.

```
1122                    // ------------------------------------------------
1123                    // ##                Shark Management              ##
1124                    // ------------------------------------------------
1125                    if (iOldValue == iMarkShark)
1126                    {
1127                        // Cache Values
1128                        y=i;
1129                        x=j;
1130
1131                        if (iDebugging == 1 && (mpiIMe == iDebugProcess ||
                                iDebugProcess == -1))
1132                            printf("Found a Shark (%d/%d)\n",x,y);
```

main.c

```
1134                        if (checkIfFishAround(aField, aBorderTop,
                                aBorderRight, aBorderBottom, aBorderLeft, &y, &x
                                ) == 1) {
1135                            /*
```

```
1136                                 # Fish around!
1137                                 */
1138
1139                                 // Over here we get 2 Sharks processed at the
                                        end of the if-Term so we need another
                                        Incriesing of iAmountProcessed
1140                                 iAmountProcessed++;
1141
1142                                 if (iDebugging == 1 && (mpiIMe == iDebugProcess
                                        || iDebugProcess == -1))
1143                                     printf("Fish around\n");
```

main.c

Die folgende Prüfung findet häufig in unserm Programm statt: Hier geht es darum ob die neue Position im aktuellen Prozess liegt oder nicht. Nur wenn dies der Fall ist kann die Bewegung lokal ausgeführt werden, sonst muss das alte Feld geleert werden und durch MPI Kommunikation der Hai in den andern Prozess verschoben werden.

```
1238                                 if (checkIfOtherProcess(y, x)==0) {
1239                                     // Move Shark to the fish's position
1240                                     moveValues(aField,i,j,y,x);
1241                                 } else {
1242                                     resetSector(aField, i, j);
1243                                 }
```

main.c

An diesem Punkt hat der Hai den Fisch gefressen und nun die Möglichkeit sich zu vermehren. Also wird im Umfeld der neuen Position des Hais nach einem freien Feld gesucht.

```
1248                                 iAmountFish--; // Decrease Amount of fishs
1249
1250                                 iRnds = 0;
1251                                 // Get position for NEW shark
1252                                 do {
1253                                     rndMove(&y, &x);
1254                                     iRnds++;
1255                                 } while ((iRnds < iRndToZero) && ( (x!=0) || (y
                                        !=0) ) && checkIfFree(aField, aBorderTop,
                                        aBorderRight, aBorderBottom, aBorderLeft, i+
                                        y, j+x)==0 );
1256
1257                                 if (iRnds < iRndToZero)
1258                                 {
```

main.c

Nun wird erneut überprüft, ob die neue Position für den Hai in dem aktuellen Prozess liegt. Wenn dies der Fall ist, dann wird an die Position ein Hai gesetzt.

```
1312                                     // Only perform setShark if not in other
                                            Process!
1313                                     if (checkIfOtherProcess(i+y, j+x)==0) {
1314                                         setShark(aField, i+y, j+x,
                                                iIntervalCount);
```

```
1315                                    }
```

main.c

Dann folgt der else Teil der If-Anweisung, wenn kein Fisch verfügbar ist. Dann bewegt sich der Hai in eine zufällige Richtung.

```
1317                                } else {
1318                                    /*
1319                                     # No Fish around -> Normal Movement
1320                                     */
1321
1322                                    if (iDebugging == 1 && (mpiIMe == iDebugProcess
                                            || iDebugProcess == -1))
1323                                        printf("No Fish around\n");
1324
1325                                    iRnds=0;
1326                                    // Get new position for shark
1327                                    do {
1328                                        rndMove(&y, &x);
1329                                        iRnds++;
1330                                    } while ( (iRnds < iRndToZero) && ( (x!=0) || (
                                        y!=0) ) && checkIfFree(aField, aBorderTop,
                                        aBorderRight, aBorderBottom, aBorderLeft, i+
                                        y, j+x)==0);
1331
1332                                    if (iRnds >= iRndToZero)
1333                                    {
1334                                        x=0;
1335                                        y=0;
1336                                    }
```

main.c

2.2.2 Fische

Im Anschluss an die Haie wird ein möglicher Fisch bearbeitet. Für diesen wird eine neue freie Position in der Umgebung gesucht. Sollte die maximale Anzahl der Schleifendurchläufe erreicht werden, muss die relative x und y Bewegung je auf 0 gesetzt werden.

```
1445                            // ------------------------------------------------
1446                            // ##                Fish Management              ##
1447                            // ------------------------------------------------
1448                            } else if (iOldValue == iMarkFish) {
1449
1450                                if (iDebugging == 1 && (mpiIMe == iDebugProcess ||
                                    iDebugProcess == -1))
1451                                    printf("Found Fish (%d/%d)\n",j,i);
1452
1453                                iRnds=0;
1454                                do {
1455                                    rndMove(&y, &x);
1456                                    iRnds++;
```

```
1457                    } while ( (iRnds < iRndToZero) && ( (x!=0) || (y
                            !=0) ) && checkIfFree(aField, aBorderTop,
                            aBorderRight, aBorderBottom, aBorderLeft, i+y, j
                            +x)==0);
1458
1459                    if (iRnds >= iRndToZero)
1460                    {
1461                        x=0;
1462                        y=0;
1463                    }
```

main.c

```
1561                    if (checkIfOtherProcess(i+y, j+x)==0) {
1562                        moveValues(aField,i,j,i+y,j+x);
1563                    } else {
1564                        resetSector(aField, i, j);
1565                    }
```

main.c

2.2.3 Plankton

Da sich Plankton ausschließlich vermehrt, wenn der Vermehrungszeitpunk aField[i][j][1] erreicht wurde, geht es hier darum in der direkten Umgebung eine freie Position zu finden. Auch hier kann es vorkommen, dass alle Positionen um das Planton belegt sind. Dann ist es notwendig, dass kein neues Planton erstellt wird. Dies ist mit der Bedingung (x!=0) || (y!=0) abgedeckt.

```
1570                    // ------------------------------------------------
1571                    // ##                Plankton Management          ##
1572                    // ------------------------------------------------
1573                    } else if ( (iOldValue == iMarkPlankton) && (aField[i][
                            j][1] == iIntervalCount) ) {
1574
1575                        if (iDebugging == 1 && (mpiIMe == iDebugProcess ||
                                iDebugProcess == -1))
1576                            printf("Found Plankton and Plankton should be
                                    respawned (%d/%d)\n",j,i);
1577
1578                        y=i;
1579                        x=j;
1580
1581                        iRnds=0;
1582                        // Get Position for new Plankton around
1583                        do {
1584                            rndMove(&y, &x);
1585                            iRnds++;
1586                        } while ( (iRnds < iRndToZero) && ( (x!=0) || (y
                                !=0) ) && checkIfFree(aField, aBorderTop,
                                aBorderRight, aBorderBottom, aBorderLeft, i+y, j
                                +x)==0);
```

```
1587
1588                    if (iRnds >= iRndToZero)
1589                    {
1590                        x=0;
1591                        y=0;
1592                    }
1593
1594                    if (iDebugging == 1 && (mpiIMe == iDebugProcess ||
                        iDebugProcess == -1))
1595                        printf("Found valid new Position (%d/%d)\n",i+y
                            ,j+x);
1596
1597                    // If no new Plankton will be set ignore the rest
                        of the procedure;
1598                    // It's for the case if Plankton is surrounded by
                        other Elements
1599                    if ( (x!=0) || (y!=0) )
1600                    {
```

main.c

```
1596                        if (checkIfOtherProcess(i+y, j+x)==0) {
1597                            // Set new Plankton
1598                            setPlankton(aField, (i+y), (j+x),
                                iIntervalCount);
1599                        }
1600                    }
```

main.c

2.3 MPI Kommunikation

Für die Parallelisierung haben wir das gesamte Simulationsfeld in einzelne rechteckige Abschnitte aufgeteilt. Die Größe der Rechtecke ist für alle Prozesse gleich groß, da die Größe für jeden Prozess fest als Konstante definiert wird.

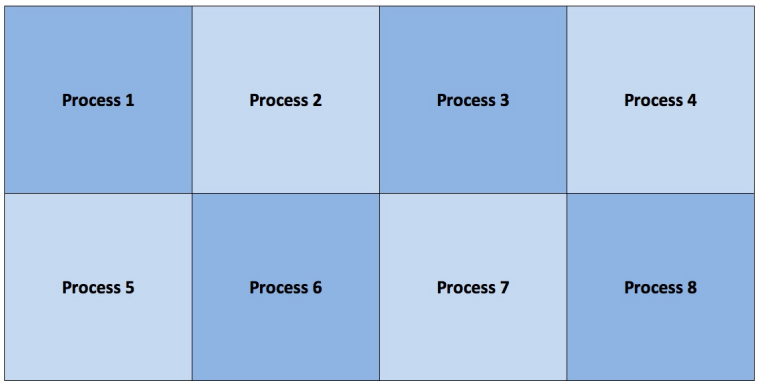

Abbildung 2.2: Beispielanordnung für 8 Prozesse

Der Programmteil, der die Ermittlung der Nachbarprozesse übernimmt wird im Abschnitt 2.3.2 „Ermittlung der Nachbarprozesse“, S. 17 erläutert.

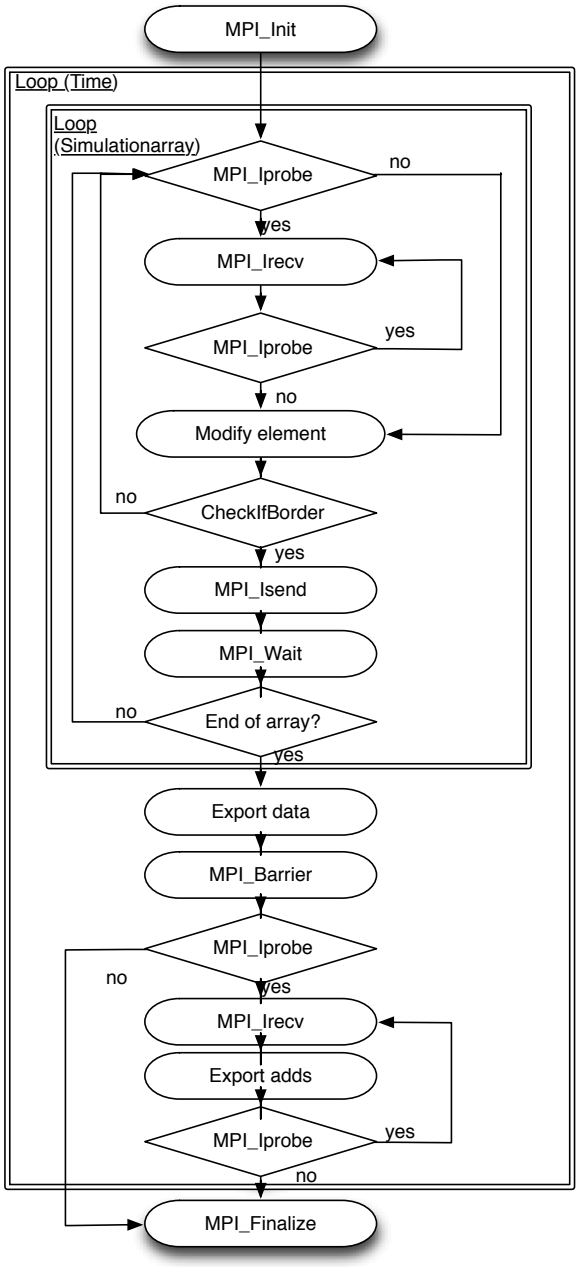

Abbildung 2.3: Kommunikationsmodell

Wir nutzen nur nicht blockierende Kommunikation für den Datenaustausch zwischen den Prozessen. Zu Beginn wird MPI inistialisiert. Dann tritt das Programm nach einigen Vorbereitungen in die Schleifen ein. Hier wird jeweils für jeden der 4 Nachbarprozesse geprüft, ob neue Bewegungsnachrichten verfügbar sind. Dies erfolgt in einer Schleife, sodass beliebig viele Nachrichten von dem jeweiligen Prozess empfangen werden können. Im Anschluss findet die Bearbeitung des einzelnen Bereichs statt. Sollte es hierbei zu einer Bewegung in den Grenzbereich des Prozesses oder in einen andern Prozess kommen, wird

dies an den jeweiligen Nachbarprozess kommuniziert. Wenn das Simulationsfeld abgearbeitet ist, schreiben alle Prozesse ihre Daten jeweils in eine Txt-Datei. Anschließend findet eine Synchronisierung aller Prozesse statt. Dann prüft jeder Prozess erneut ob es noch Nachrichten von andern Prozessen gibt. Sollten hier Daten empfangen werden, die einen Prozesswechsel darstellen, wird dies nachträglich in die Txt-Datei hinzugefügt.

2.3.1 Kommunikationsformat

Für die Kommunikation wird je ein 5 elementiger Array versendet, in dem folgende Daten übertragen werden:

1. Alte Y-Position
2. Alte X-Position
3. Neue Y-Position
4. Neue X-Position
5. Item Typ

Die Koordinaten können hier auch -2 enthalten, dies bedeutet, dass hier keine vorherige Information verfügbar ist. Zu beachten ist hier, dass die Positionen relativ zum aktuellen Prozess angegeben werden. Somit müssen diese nach dem Empfangen auf den neuen Prozess umgerechnet werden.

2.3.2 Ermittlung der Nachbarprozesse

Folgender Code ermittelt für jeden Prozess die Nachbarprozesse, mit denen kommuniziert werden muss.

Es werden maximal 4 Prozesse in einer Reihe positioniert.

```
571     // Determine how many processes per line will be initialized
572     if (mpiISize%4==0)
573         mpiIProcessPerline=4;
574     else if (mpiISize%3==0)
575         mpiIProcessPerline=3;
576     else if (mpiISize%2==0)
577         mpiIProcessPerline=2;
578     else if (mpiISize%1==0)
579         mpiIProcessPerline=1;
580
581     // Debugging Processes per line
582     if (mpiIMe==iDebugProcess && iDebugging>0)
583         printf("I will place %i Processes per line \n\n",
                mpiIProcessPerline);
584
585
586     // ------------------------------------------------
```

```
587     // ## Determine Neighborhood for each Process ##
588     // ---------------------------------------------
589
590     // Determine Neighbors
591     //
592     // Visualisaton 12 Processes
593     // |-|-|-|-|
594     // |0|1|2|3|
595     // |-|-|-|-|
596     // |4|5|6|7|
597     // |-|-|-|-|
598     // |8|9|10|11|
599     // |-|-|-|-|
600     //
601
602     // TOP
603     if (0==(mpiIMe/mpiIProcessPerline)) // Bottom & Rest
604         mpiITop=((mpiISize/mpiIProcessPerline)-1)*mpiIProcessPerline+(
                mpiIMe%mpiIProcessPerline);
605     else if ((mpiISize/mpiIProcessPerline)>(mpiIMe/mpiIProcessPerline)) //
            Top
606         mpiITop=mpiIMe-mpiIProcessPerline;
607
608     // Right
609     if ((mpiIMe%mpiIProcessPerline)<(mpiIProcessPerline-1))
610         mpiIRight=(mpiIMe+1);
611     else
612         mpiIRight=(mpiIMe-(mpiIProcessPerline-1));
613
614     // Bottom
615     if (((mpiISize/mpiIProcessPerline)-1)==(mpiIMe/mpiIProcessPerline)) //
            Bottom
616         mpiIBottom=(mpiIMe%mpiIProcessPerline);
617     else if ((mpiISize/mpiIProcessPerline)>(mpiIMe/mpiIProcessPerline)) //
            Top & Rest
618         mpiIBottom=mpiIMe+mpiIProcessPerline;
619
620     // Left
621     if ((mpiIMe%mpiIProcessPerline)>0)
622         mpiILeft=(mpiIMe-1);
623     else
624         mpiILeft=(mpiIMe+(mpiIProcessPerline-1));
```

main.c

2.3.3 Versenden der Bewegungsnachrichten

Es gibt drei Fälle in denen eine Bewegung an einen Nachbarprozess kommuniziert werden muss.

1. Ein Item wird in den äußersten Bereich (Grenzbereich) eines Prozesses gesetzt oder bewegt

2. Ein Item wird aus dem Grenzbereich in einen **andern** Prozess bewegt

3. Ein Item wird aus dem Grenzbereich in den **gleichen** Prozess bewegt

Die ersten zwei Fälle werden mit folgender Bedingung geprüft, die Funktion checkIfBorder(y,x) gibt jeweils die Richtung der Kommunikation oder 0 für keinen Grenzbereich oder Prozesswechsel aus.

```
1145        if (checkIfBorder(y, x)>0)
1146        {
```

main.c

Der letzte Fall ist dann eine Umkehrung der vorherigen Bedingung.

```
1197  // Values leaving the Border
1198  if (checkIfBorder(i, j)>0 && checkIfBorder(i+y, j+x)==0)
1199  {
```

main.c

Wenn jeweils eine Bedingung erfüllt ist, wird zuerst der zu versendende Array vorbereitet. Außerdem wird eine Variable temp mit der notwendigen Richtung des Versand gefüllt und eine Variable bOtherProcess mit einem Wahrheitswert, der angibt, ob die aktuelle Bewegung den Prozess verlässt. In letzerem Fall muss sofort der entsprechende Wert in dem Grenzbereich Array der entsprechenden Richtung gesetzt werden.

```
1150  int temp = checkIfBorder(i+y, j+x);
1151  int bOtherProcess = checkIfOtherProcess(y, x);
1152  mpiAData[0] = i;
1153  mpiAData[1] = j;
1154  mpiAData[2] = y;
1155  mpiAData[3] = x;
1156  mpiAData[4] = iMarkShark;
```

main.c

Anschließend findet der eigentliche Versand in eine der vier möglichen Richtungen statt. Wichtig in diesem Zusammenhang ist die Verwendung der MPI_Wait Funktion, da die versendeten Daten bis zu Abschluss des Versands nicht verändert werden dürfen.

```
1158                    // Sent Data to the Process above
1159                    if (temp == 1)
1160                    {
1161                        MPI_Isend(mpiAData, 5, MPI_INT ,
                                mpiITop, 0, MPI_COMM_WORLD, &
                                mpiRRequest);
1162                        // If Element moves to other Process;
                                Element in Border needs to be
                                created
1163                        if (bOtherProcess > 0)
1164                            aBorderTop[x]=iMarkShark;
1165                        MPI_Wait(&mpiRRequest,&mpiStat);
1166                    }
1167
```

```
1168                    if (temp == 2)
1169                    {
1170                        MPI_Isend(mpiAData, 5, MPI_INT ,
                                mpiIRight, 0, MPI_COMM_WORLD, &
                                mpiRRequest);
1171                        // If Element moves to other Process;
                                Element in Border needs to be
                                created
1172                        if (bOtherProcess > 0)
1173                            aBorderRight[y]=iMarkShark;
1174                        MPI_Wait(&mpiRRequest,&mpiStat);
1175                    }
1176
1177                    if (temp == 3)
1178                    {
1179                        MPI_Isend(mpiAData, 5, MPI_INT ,
                                mpiIBottom, 0, MPI_COMM_WORLD, &
                                mpiRRequest);
1180                        // If Element moves to other Process;
                                Element in Border needs to be
                                created
1181                        if (bOtherProcess > 0)
1182                            aBorderBottom[x]=iMarkShark;
1183                        MPI_Wait(&mpiRRequest,&mpiStat);
1184                    }
1185
1186                    if (temp == 4)
1187                    {
1188                        MPI_Isend(mpiAData, 5, MPI_INT ,
                                mpiILeft, 0, MPI_COMM_WORLD, &
                                mpiRRequest);
1189                        // If Element moves to other Process;
                                Element in Border needs to be
                                created
1190                        if (bOtherProcess > 0)
1191                            aBorderLeft[y]=iMarkShark;
1192                        MPI_Wait(&mpiRRequest,&mpiStat);
1193                    }
```

main.c

2.3.4 Empfangen der Bewegungsnachrichten

Der folgende Programmcode prüft für alle Richtungen, ob eine Nachricht verfügbar ist und empfängt diese. Anschließend wird der übertragene Datensatz an die Funktion specialMovement übergeben. Diese Funktion wird im Abschnitt 2.3.5 „Verarbeiten der Bewegungsnachricht“, S. 21 erläutert.

```
984                 // MPI: Check if another Process wants to transmit a Border
                        -Movement
```

```
985                 MPI_Iprobe(mpiITop, 0, MPI_COMM_WORLD, &mpiIBorderTouched,
                        &mpiStat);
986
987                 // If there is Data to recieve => Recieve it
988                 while (mpiIBorderTouched == 1) {
989                     if (iDebugging == 2 && (mpiIMe == iDebugProcess ||
                            iDebugProcess == -1))
990                         printf("Received from Top\n");
991
992                     // Recieve Stuff
993                     MPI_Irecv(mpiAData, 5, MPI_INT, mpiITop, 0,
                            MPI_COMM_WORLD, &mpiRRequest);
994                     specialMovement(aField, aBorderTop, aBorderRight,
                            aBorderBottom, aBorderLeft, mpiAData, 1,
                            iIntervalCount);
```

main.c

2.3.5 Verarbeiten der Bewegungsnachricht

Es existieren grundsätzlich 3 Bewegungsszenarien, die bereits in Abschnitt 2.3.3 „Versenden der Bewegungsnachrichten“, S. 18 erläutert wurden.
Für die Verarbeitung einer Nachricht müssen anhand des Versenders und der Bewegungsart die notwendigen Aktionen ermittelt werden.

```
364 // Function to deal with Border-Movement and Objects moving to other
        Process
365 //
366 //
367 void specialMovement(int ***array, int *aBorderTop, int *aBorderRight, int
        *aBorderBottom, int *aBorderLeft, int *data, int from, int actual)
368 {
369
370     // Specification for from:
371     // 1 = Top
372     // 2 = Right
373     // 3 = Bottom
374     // 4 = Left
375
376     // Specification for Data Array:
377     // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
378     // !! Be Carful: Position is from Process sending the Data !!
379     // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
380     // 0 = OLD-Y
381     // 1 = OLD-X
382     // 2 = NEW-Y
383     // 3 = NEW-X
384     // 4 = Type
385
386     if (iDebugging == 2)
387         printf("Special Movement: %d/%d/%d/%d/%d/%d\n", from, data[0], data
            [1], data[2], data[3], data[4]);
```

```
388
389     // *TOP*
390     if (from == 3)
391     {
392         // Border Area
393         if (data[2] == (iFieldSizeY-1))
394         {
395             // Check if Items has been in Border before!!
396             if (data[0] == (iFieldSizeY-1))
397             {
398                 aBorderTop[(data[1])] = 0;
399             }
400             aBorderTop[(data[3])] = data[4];
401         // Process Change
402         } else if (data[2] == iFieldSizeY)
403         {
404             if (data[0] == (iFieldSizeY-1))
405             {
406                 aBorderTop[(data[1])] = 0;
407             }
408             setValue(array, 0, data[3], data[4], actual);
409         } else if (data[2] == -2)
410         {
411             if (data[0] == (iFieldSizeY-1))
412             {
413                 aBorderTop[(data[1])] = 0;
414             }
415         }
416     } else if (from == 4)
417     {
418         // Border Area
419         if (data[3] == (iFieldSizeX-1))
420         {
421             // Check if Items has been in Border before!!
422             if (data[1] == (iFieldSizeX-1))
423             {
424                 aBorderRight[(data[0])] = 0;
425             }
426             aBorderRight[(data[2])] = data[4];
427             // Process Change
428         } else if (data[3] == iFieldSizeX)
429         {
430             if (data[1] == (iFieldSizeX-1))
431             {
432                 aBorderRight[(data[0])] = 0;
433             }
434             setValue(array, data[2], 0, data[4], actual);
435         } else if (data[3] == -2)
436         {
437             if (data[1] == (iFieldSizeX-1))
438             {
439                 aBorderRight[(data[0])] = 0;
440             }
```

```
441         }
442     } else if (from == 1)
443     {
444         // Border Area
445         if (data[2] == 0)
446         {
447             // Check if Items has been in Border before!!
448             if (data[0] == 0)
449             {
450                 aBorderBottom[(data[1])] = 0;
451             }
452             aBorderBottom[(data[3])] = data[4];
453             // Process Change
454         } else if (data[2] == -1)
455         {
456             if (data[0] == 0)
457             {
458                 aBorderBottom[(data[1])] = 0;
459             }
460             setValue(array, 0, data[3], data[4], actual);
461         } else if (data[2] == -2)
462         {
463             if (data[0] == 0)
464             {
465                 aBorderBottom[(data[1])] = 0;
466             }
467         }
468     } else if (from == 2)
469     {
470         // Border Area
471         if (data[3] == 0)
472         {
473             // Check if Items has been in Border before!!
474             if (data[1] == 0)
475             {
476                 aBorderLeft[(data[0])] = 0;
477             }
478             aBorderLeft[(data[2])] = data[4];
479             // Process Change
480         } else if (data[3] == -1)
481         {
482             if (data[1] == 0)
483             {
484                 aBorderLeft[(data[0])] = 0;
485             }
486             setValue(array, data[2], 0, data[4], actual);
487         } else if (data[3] == -2)
488         {
489             if (data[1] == 0)
490             {
491                 aBorderLeft[(data[0])] = 0;
492             }
493         }
```

```
494     }
495 }
```

main.c

2.4 Export in Datei

Der Export in die Txt-Datei erfolgt mit Hilfe der folgenden zwei Funktionen (ausgelagert in debug.h):

```
39 void writeArray(int iAmount, int iProcess, int*** array, int y, int x, int
       noProcesses) {
40
41     // Build File-Name
42     char buffer2 [255];
43
44     int iFirstValue=0;
45
46     sprintf(buffer2,"/Users/Alexander 1/Downloads/FishShark/%03i-%03i.txt",
           iAmount,iProcess);
47     //sprintf(buffer2,"/home/lambertz/FishShark/data/%03i-%03i.txt",iAmount
           ,iProcess);
48
49     FILE *file;
50     file = fopen(buffer2,"a");
51
52     fprintf(file,"NoProcesses:%d,Y:%d,X:%d;\n", noProcesses, y, x);
53
54     if (iOutputFormat == 1)
55     {
56         for (int i=0; i<y; i++) {
57             if (i!=0)
58                 fprintf(file,"\n");
59             for (int j=0; j<x; j++) {
60                 if (array[i][j][0]==0)
61                     fprintf(file,";");
62                 else
63                     fprintf(file,"%d;",array[i][j][0]);
64             }
65         }
66     } else if (iOutputFormat == 2) {
67         for (int i=0; i<y; i++) {
68             if (i!=0)
69                 fprintf(file,"\n");
70             for (int j=0; j<x; j++) {
71                 if (array[i][j][0]!=0)
72                 {
73                     if (iFirstValue == 0)
74                     {
75                         fprintf(file,"%d:%d", j, array[i][j][0]);
76                         iFirstValue=1;
77                     } else {
```

```
78                          fprintf(file,",%d:%d", j, array[i][j][0]);
79                      }
80                  }
81              }
82              fprintf(file,";");
83              iFirstValue=0;
84          }
85      }
86
87      fprintf(file,"\n");
88
89      //fprintf(file,"\n\n");
90      fclose(file);
91  }
```

debug.h

Die erste Funktion schreibt den Inhalt des Arrays in eine Txt-Datei. Während der Entwicklung haben wir das Format des Exports verändert. Beide Formate sind stets verfügbar. Das verwendete Format wird durch eine Konstante zu Beginn der Datei eingestellt.

Die Dateiformate unterscheiden sich im wesentlichen darin, dass die Datei im ersten Dateiformat eine 1:1 Abbildung des Arrays ist. So stellt die jeweilige Zeile die Y-Koordinate dar und der jeweilige Wert zwischen den Semikola bildet das Element ab. In der hier implementierten Version werden keine 0-Werte in die Datei geschrieben. Dies spart Speicherplatz.

Jedoch besonders bei sehr leeren Simulationsfeldern kommt es zu dem Problem, dass das erste Export-Format sehr groß ist. Somit haben wir eine leicht abgewandelte Version entwickelt. Diese Version sieht vor, dass die Zeilen erneut der Y-Koorinate entsprechen, jedoch aus der X-Achse nur Werte exportiert werden, die ungleich 0 sind und dann vor ihnen die jeweilige X-Koordinate ausgegeben wird.

Z.B. aField[3][10][0] = 1 wird wie folgt abgebildet:

```
1
2  10:1;
```

Mit der nachfolgenden Funktion werden Daten, die nach der Barriere empfangen werden, an die Txt-Datei angehängt. Hier werden die Daten in folgendem Format übertragen $< Y - Position >, < X - Position >: < Element >;$.

```
93  void addToOutput(int iAmount, int iProcess, int x, int y, int value)
94  {
95      // Build File-Name
96      char buffer2 [255];
97
98      sprintf(buffer2,"/Users/Alexander 1/Downloads/FishShark/%03i-%03i.txt",
            iAmount,iProcess);
```

```
99      //sprintf(buffer2,"/home/lambertz/FishShark/data/%03i-%03i.txt",iAmount
            ,iProcess);
100
101
102     FILE *file;
103     file = fopen(buffer2,"a");
104
105     fprintf(file,"%d,%d:%d;", x, y, value);
106
107     fclose(file);
108
109 }
```

debug.h

2.5 Visualisierung

Da erste Entwürfe einer parallelen Visualisierung (im direkten Zusammenhang mit der Simulation) zu vielen Problemen führten, entschieden wir uns, das Hauptaugenmerk auf die Simulation zu richten. Dementsprechend wurde die Zwischenspeicherung der Daten in Dateien, sowie eine sequentielle und von der Simulation getrennte Visualisierung gewählt.

Zunächst wird mithilfe der dirent.h der Unterordner "./data", in dem sich die Daten befinden, geöffnet. Anschließend werden über eine Schleife alle vorhandenen Dateien eingelesen. Dies muss in alphabetischer Reihenfolge stattfinden. Gelöst wurde das zunächst per readdir(). Da hier die alphabetische Sortierung unter Windows und MacOS nur zufällig erfolgte, unter Unix die Dateien aber in zufälliger Reihenfolge eingelesen wurden, wurde das readdir() durch ein scandir() ersetzt.

Ist die Datei geöffnet, werden verschiedene Variablen bereitgestellt, darunter beispielsweise ein Zähler für die Zeilen, ein Zähler für die Anzahl gelesener Zeichen, ein Zähler für die Anzahl der aus der ersten Zeile gelesenen Daten, sowie einige Integer, die als boolean fungieren und festhalten, ob man sich beispielsweise in der ersten Zeile befindet oder ob gerade eine neue Zeile angefangen wurde. Außerdem wird noch ein struct bmpData genutzt, das immer einen gelesenen Datensatz speichert.

```
46 struct bmpData{
47     char posX[6];
48     char posY[6];
49     char val[2];
50     int isPos;
51     int isX;
52 };
```

visual_main.c

Dieser beinhaltet:

- posX: Die X-Koordinate des Datensatzes
- posY: Die Y-Koordinate des Datensatzes (wird nur bei den Zusatzdaten benötigt, dazu später)
- val: Der Wert für Fische (1), Haie (2) oder Plankton (3)
- isPos: Bei 1 wird eine Position gelesen, bei 0 wird val gelesen
- isX: Bei 1 wird der X-, bei 0 wird der Y-Achsen-Wert gelesen

Beim Einlesen der Datei, das immer zeichenweise geschieht, wird nun immer wie folgt vorgegangen:

```
141                 if( iFirstValues < 3 && ( ( 0 == iFileCounter ) || ( 0
                        == iFileCounter % iProcess ) ) ){
142                     //Nur jede iProzess-te Datei darf ein neues bmp
                            angefangen werden
143                     // die Grund-Variablen veraendert werden
144                     // in der ersten Zeile muessen dann spezielle Daten
                            ausgelesen werden
145                     if( ( ';' == c ) || ( ',' == c ) ){
146                         //Variable kann "gefuellt" werden
147                         if( 0 == iFirstValues ){
148                             sscanf(cFirstRow, "%d", &iProcess);
149                             if( DEBUG ) printf("VARIABLE iProcess = %d\
                                    n", iProcess);
150                         }
151                         else if( 1 == iFirstValues ){
152                             sscanf(cFirstRow, "%d", &iSizeY);
153                             if( DEBUG ) printf("VARIABLE iSizeY = %d\n"
                                    , iSizeY);
154                         }
155                         else if( 2 == iFirstValues ){
156                             sscanf(cFirstRow, "%d", &iSizeX);
157                             if( DEBUG ) printf("VARIABLE iSizeX = %d\n"
                                    , iSizeX);
158                         }
159                         memset(cFirstRow, '\0', 6);
160                         iFirstValues++;
161
162                         if( 3 == iFirstValues ){
163                             if( DEBUG ) printf("Neues bmp wird erstellt
                                    ...\n");
164                             bmp = bmp_create((iPixelSize * iSizeX * 4),
                                     (iPixelSize * iSizeY * (iProcess/4)),
                                    24);
165                             if( DEBUG ) printf("Neues bmp erstellt.\n")
                                    ;
166                             iFirstRow = 0;
167                         }
168                     }
```

```
169                     else{
170                         strncat(cFirstRow, &c, 1);
171                         if( DEBUG_EXT ) printf("first row: %s\n",
                                cFirstRow);
172                     }
173                     if( DEBUG_EXT ) printf("VARIABLE iFirstValues: %d\n
                            ", iFirstValues);
174                 } //if(erste Zeile vom ersten Abschnitt)
```

visual_main.c

Die erste Zeile einer Datei beinhaltet immer die gleichen Daten:

- Die Anzahl beteiligter Prozesse pro Bild (Simulationsschritt)
- Die Größe des Spielfeldes auf der Y-Achse
- Die Größe des Spielfeldes auf der X-Achse

Diese erste Zeile darf nur bei der jeweils ersten Datei eines jeden Bildes gelesen werden.

Es wird nun zeichenweise gelesen und jedes Zeichen einem String angehängt. Wird ein Kommata (Datensatz zu Ende) oder ein Semikolon (Zeile zu Ende) gelesen, muss dieser String nun als spezieller Datensatz gespeichert werden. Dabei wird an einem Zähler überprüft, wie viele Datensätze schon gelesen wurden und welcher daraus resultierend nun folgen muss. Anschließend wird der String wieder geleert und für den nächsten Datensatz verfügbar gemacht. Wurden alle drei speziellen Datensätze eingelesen, kann das Bild für die folgenden Daten erzeugt werden.

```
175                 else if( iFirstRow ){
176                     //erste Zeile darf ansonsten nicht als Pixel
                            interpretiert werden!
177                     if( ';' == c ) iFirstRow = 0;
178                 }
```

visual_main.c

Dieser Code-Ausschnitt handhabt die Situation, dass die erste Zeile gelesen wird, es sich bei der Datei aber nicht um die erste eines Bildes handelt. Die bis zum ersten Semikolon gelesenen Zeichen müssen ignoriert werden, da sie nicht als eigentlicher Datensatz interpretiert werden dürfen/können.

```
218                 else{
219                     //ansonsten normale Fallunterscheidung fuer iSizeY
                            Zeilen
220                     if( ':' == c ){
221                         //Wert folgt
222                         data.isPos = 0;
223                     }
224                     else if( ',' == c  || ';' == c){
225                         //neuer Datensatz bzw...
226                         if( 0 == iNewRow){
```

```
227                                 data.isPos = 1;
228                                 int iMomentaryProcess = iFileCounter %
                                        iProcess;
229                                 mySetPixel(bmp, atoi(data.val), atoi(data.
                                        posX), iRowCounter, iMomentaryProcess,
                                        iSizeX, iSizeY);
230                                 memset(data.posX, '\0', sizeof(data.posX)/
                                        sizeof(char));
231                                 memset(data.val, '\0', sizeof(data.val)/
                                        sizeof(char));
232                             }
233                             else iNewRow = 0;
234                             if( ';' == c) iRowCounter++;
235                             //... neue Zeile
236                         }
237                         else if( '\n' == c || '\0' == c ){
238                             if( '\n' == c ) iNewRow = 1;
239                         }
240                         else{
241                             if( data.isPos ){
242                                 strncat(data.posX, &c, 1);
243                             }
244                             else{
245                                 strncat(data.val, &c, 1);
246                             }
247
248                             //falls ein Zeichen zuvor eine neue Zeile
                                    angefangen wurde:
249                             iNewRow = 0;
250                         }
251                     }
```

visual_main.c

Als nächstes wird der "Normalfall"behandelt: Bis zu einem Trennungszeichen (Semikolon oder Kommata) werden Zeichen entweder als X-Koordinate oder als Wert interpretiert und dementsprechend im bmpData-struct "verstaut". Beim Lesen eines der beiden Trennungszeichen zählt ein Datensatz als abgeschlossen und kann somit als Pixel in der bmp-Datei eingetragen werden (Aufruf mySetPixel()). Hierfür werden unter anderem X- und Y-Koordinate (letztere als Zeilenzähler), der Wert, der momentane "Teilprozess", sowie die Teilozeangröße übergeben.

```
54 rgb_pixel_t myGetColor(int value){
55     if( DEBUG_EXT ) printf("COLOR should be %d.\n", value);
56     if(value == FISH){
57         return pFish;
58     }
59     if(value == SHARK){
60         return pShark;
61     }
62     if(value == PLANKTON){
63         return pPlankton;
```

```
64      }
65      rgb_pixel_t pError = {255, 0, 255, 0};
66      if( DEBUG_SIMPLE ) iErrorCounter++;
67      return pError;
68  }
69
70  void mySetPixel(bmpfile_t *bmp, int iValue, int dx, int dy, int
        iMomentaryProcess, int iSizeX, int iSizeY){
71      rgb_pixel_t pColor = myGetColor(iValue);
72      int x = (iSizeX * ( iMomentaryProcess % 4 ) ) + dx;
73      int y = (iSizeY * ( iMomentaryProcess / 4 ) ) + dy;
74      bmp_set_pixel(bmp, x, y, pColor);
75      if( DEBUG_EXT ) printf("PIXEL at p(%d),l(%d|%d),g(%d|%d) set to %d.\n",
            iMomentaryProcess, dx, dy, x, y, iValue);
76      if(( iValue == 0 || iValue > 3) && ( DEBUG_SHOW_PIXEL_ERRORS )) printf(
            "ERROR: Pixel at p(%d),l(%d|%d),g(%d|%d) set to %d.\n",
            iMomentaryProcess, dx, dy, x, y, iValue);
77  }
```

visual_main.c

Durch einen Aufruf von myGetColor() wird die Farbe ermittelt, die der Pixel haben soll. Fische werden blau, Haie rot und Plankton grün eingefärbt. Außerdem muss aus den als Parametern übergebenen Werten noch die globale Position des Pixels in der Bilddatei ermittelt werden, da die übergebenen Daten sich auf die jeweiligen Teilozeane beziehen. Nachdem alle regulären Daten eingelesen wurden, kann es allerdings sein, dass noch einige Datensätze (diesmal mit Y-Koordinate) an die Datei angehängt wurden. Diese sind im Format "X-Koordinate, Y-Koordinate: Wert"gespeichert und müssen wie folgt behandelt werden:

```
179                     else if( iRowCounter >= iSizeY ){
180                         //zusaetzliche Informationen, nachdem alle Zeilen
                                eingelesen wurden
181                         if( ':' == c){
182                             //Wert folgt
183                             data.isPos = 0;
184                         }
185                         else if( ',' == c ){
186                             data.isX = 0;
187                         }
188                         else if( ';' == c ){
189                             //Pixel setzen
190                             int iMomentaryProcess = iFileCounter % iProcess
                                    ;
191                             //if( DEBUG_SHOW_PIXEL_ERRORS && ( (atoi(data.
                                    val) == 0) || ( atoi(data.val) > 3 ) ) )
                                    printf("ERROR aus Add folgend:\n");
192                             mySetPixel(bmp, atoi(data.val), atoi(data.posX)
                                    , atoi(data.posY), iMomentaryProcess, iSizeX
                                    , iSizeY);
193                             memset(data.posX, '\0', sizeof(data.posX)/
                                    sizeof(char));
```

```
194                             memset(data.posY, '\0', sizeof(data.posY)/
                                    sizeof(char));
195                             memset(data.val, '\0', sizeof(data.val)/sizeof(
                                    char));
196                             data.isPos = 1;
197                             data.isX = 1;
198                             iRowCounter++;
199                             iActualAdds++;
200                         }
201                         else if( '\n' == c || '\0' == c ){}
202                         else{
203                             if( data.isPos ){
204                                 if( data.isX ){
205                                     //X-Wert lesen
206                                     strncat(data.posX, &c, 1);
207                                 }
208                                 else{
209                                     //Y-Wert lesen
210                                     strncat(data.posY, &c, 1);
211                                 }
212                             }
213                             else{
214                                 strncat(data.val, &c, 1);
215                             }
216                         }
217                     } //if(zusaetzliche Zeilen nach iSizeY)
```

visual_main.c

Je nach Trennungszeichen werden weitere Zeichen entsprechend im struct verstaut und anschließend erneut als Pixel interpretiert. Ist eine Datei komplett eingelesen, wird sie geschlossen und es muss gegebenenfalls das fertige Bild gespeichert werden:

```
273             if( 0 == ( iFileCounter+1 ) % iProcess ){
274                 int iChecksum = (iFileCounter+1) % iProcess;
275                 if( DEBUG_EXT ) printf("CHECKSUM: 0 =?= %d, (%d+1), %d\n",
                        iChecksum, iFileCounter, iProcess);
276                 if( DEBUG ) printf("OUTPUT Datei speichern...\n");
277                 //spezieller Fall:
278                 //bmp speichern und zerstoeren
279                 char cOutputName[30];
280                 char *cOutputFolder = "./output/";
281                 int iPic = ( iFileCounter + 1 ) / iProcess;
282
283                 sprintf(cOutputName, "%s%d%s", cOutputFolder, iPic, ".bmp")
                        ;
284
285                 bmp_save(bmp, cOutputName);
286                 bmp_destroy(bmp);
287                 if( DEBUG_SIMPLE ) printf("OUTPUT %s gespeichert.\n",
                        cOutputName);
288                 if( DEBUG_SIMPLE ) printf("
                        ==========\n\n");
289             }
```

visual_main.c

Wie den Code-Beispielen zu entnehmen ist, ist die Visualisierung mit zahlreichen Debug-Informationen versehen, die nach Bedarf eingestellt werden können:

```
27 const int DEBUG_SIMPLE = 1;       //Nur fuer die noetigsten Debug Outputs
28 const int DEBUG = 0;               //Debug Output einblenden
29 const int DEBUG_EXT = 0;           //Erweiterte Debug Outputs einblenden
30 const int DEBUG_SHOW_CHARS = 0; //Ausgabe jedes gelesenen Chars
31 const int DEBUG_SHOW_BMPDATA = 0; //Ausgabe des structs
32 const int DEBUG_SHOW_PIXEL_ERRORS = 1;       //Fuer Error-Ausgaben bzgl.
```

visual_main.c

- DEBUG_SIMPLE zeigt nur gelesene Dateinamen sowie gespeicherte bmp-Dateien an.
- DEBUG gibt pro Datei gelesene Zeichen und Zeilen aus.
- DEBUG_EXT(ENDED) zeigt unter anderem jede Pixelfärbung an.
- DEBUG_SHOW_CHARS gibt jedes gelesene Zeichen einzeln aus.
- DEBUG_SHOW_BMPDATA gibt bei jedem gelesenen Zeichen relevante Werte aus dem bmpData-struct aus.
- DEBUG_SHOW_PIXEL_ERRORS zeigt einen Hinweis bezüglich falscher Pixel (meistens wenn der Wert = 0 ist). Diese werden außerdem in Rosa in der Ausgabe-Datei dargestellt.

Anmerkung: Für einen Visualisierungsdurchlauf mit vielen Datensätzen empfiehlt es sich, nur DEBUG_SIMPLE und DEBUG_SHOW_PIXEL_ERRORS zu nutzen.

3 Auswertung

Wir haben einige Zeit benötigt, um geeignete Paramter für unsere Simulation zu finden, und entschieden dannein 20.000 x 10.000 Pixel großes Simulationsfeld zu verwenden, auf dem sich 50% Fische, 5%Haie und 1% Plankton befinden. Die Simulation lief jeweils über 100 Bilder.

Aus dieser Messung ergibt sich folgender Graph:

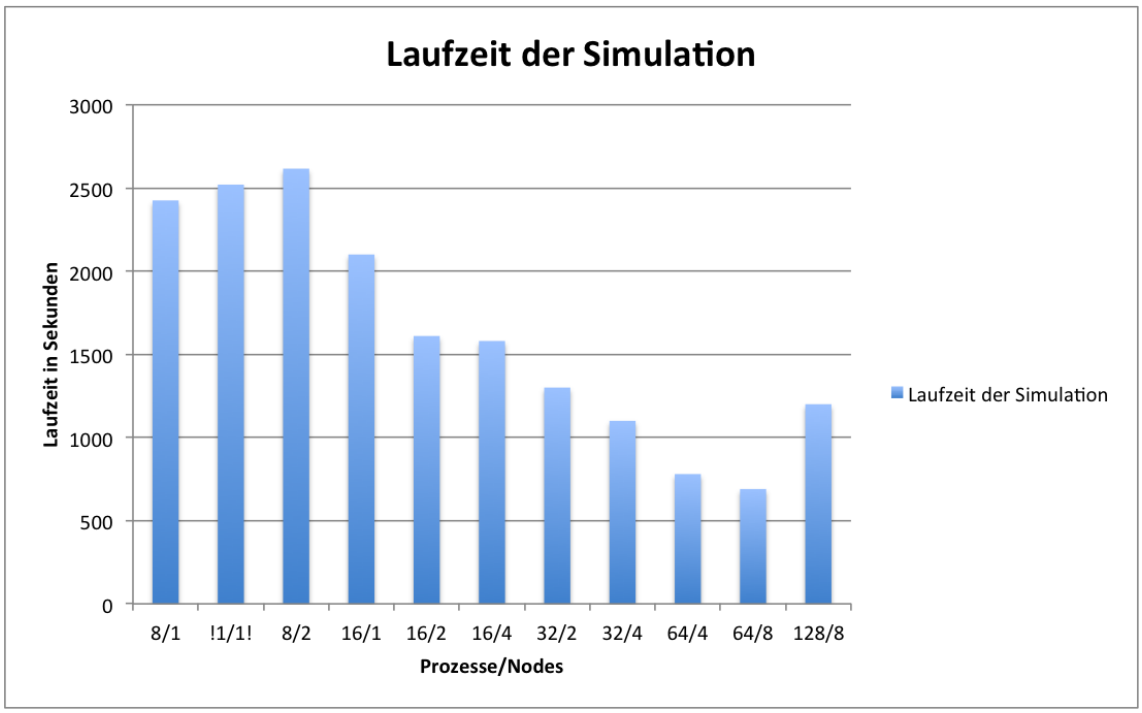

Abbildung 3.1: Laufzeit der Simulation

In diesem Graphen sind die Ausführungszeit dargestellt. Die sequentielle Ausführungszeit des Programms, ohne MPI-Kommunikation bei gleichen Parametern, lag bei 2520 Sekunden. Daraus folgt, dass bei 8 Prozessen auf 2 Notes der Kommunikations-Overhead zu groß ist, um die sequeltielle Ausführungszeit zu erreichen. Wohingegen bei 8 Prozessen auf einem Node die Zeit unterboten werden kann. der Graph fällt bis hin zu 64 Prozessen stetig ab. Anschließend ist das einzelne Simulationsfeld zu klein damit der Kommunikations-Overhead kompensiert werden kann. Somit steigt die Ausführungszeit hier wieder deutlich an.

Außerdem muss beachtet werden, dass vor allem der Umfang der Kommunikation sehr

vom Zufall beeinflusst ist. Bei den oben genannten Parametern kann es theoretisch dazu kommen, dass alle Prozesse völlig unabhängig voneinander arbeiten. So entsteht ausschließlich ein Overhead durch das Prüfen auf neue Nachrichten mit der MPI_IPROBE Funktion.

Um diese Messdaten zu erhalten haben wir jeden Simulationsschritt 4 mal durchgeführt und dann das Mittel aller Ausführungszeiten gebildet. Daraus ergeben sich auch die relativ großen Schritte, da sonst die Simulation zu lange Zeit in Anspruch genommen hätte.

Wir gestehen jedoch ein, dass bei 12 Prozessoren pro Node die Wahl der Simulationsschritte ungünstig ausfiel.
Alles in Allem sind wir mit unserer Implementation jedoch sehr zufrieden. Wir können auch große Simulationsfelder effektiver als unsere sequentielle Implementation auswerten.

4 Genutzte Software/Hardware

- Apple XCode
  Erstellung des Programmcodes; Debuggen der sequentiellen Version
  Ohne MPI Unterstützung, da dies nur mit umfangreichen Änderungen an der Software möglich gewesen wäre
- Alinea ddt auf dem Cluster
  Debuggen des parallelen Programmcodes
- valgrind
  Debugging (Auffinden von Speicherleaks)
- OmniGraffle
  Erstellung diverser Grafiken
- git
  Versionsverwaltung
- Texmaker
  Erstellung dieser Ausarbeitung
- Microsoft Powerpoint
  Erstellung der Präsentationen

5 Aufgetretene Probleme

In diesem Kapitel möchten wir nun auf Probleme eingehen, die während unserer Entwicklung aufgetreten sind.

5.1 Nutzung blockierender Kommunikation

Wir haben unsere parallele Implementation zunächst mit blockierender MPI-Kommunikation erstellt. Daraus ergaben sich jedoch Deadlocks, die nach intensiver Studie unseres Konzepts durch den Einsatz nicht blockierender MPI-Kommunikation gelöst werden konnten.

5.2 Nutzung nicht blockierender Kommunikation ohne anschließendes MPI_Wait

Nach der Umstellung unsers Programms auf nicht blockierende Kommunikation hatten wir Probleme mit Nachrichten, die uns aus mehreren Nachrichten zusammengesetzt erschienen. Nach Studium der MPI Hilfe von der TU Chemnitz[3] stießen wir darauf, dass die Daten bis zum Abschluss des Sendevorgangs nicht verändert werden dürfen und haben somit MPI_Wait Befehle in den Quelltext eingefügt.

5.3 Sunshot

Von Seiten des DKRZ wurde uns der Zugang zu Sunshot zur Verfügung gestellt. Leider war die Software wohl auf Grund von Umstrukturierungen des Clusters für uns gegen Ende nicht mehr verfügbar. Frühere Versuche scheiterten wegen zu großer Dateien am Überlaufen des Java Heap, selbst wenn nur einzelne Trace-Dateien ausgewertet werden sollten.

5.4 Lange Laufzeit der Schleifen

Bei einem sehr vollen Simulationsfeld dauert das Suchen nach einer freien Position um z.B. einen Fisch zu bewegen sehr lange. Deshalb haben wir eine Konstante für die maximalen Schleifendurchläufe implementiert.

5.5 Streifenbildung in der Visualisierung

Die Ursache für eine leichte Streifenbildung in der Visualisierung liegt darin, dass der Border-Bereich nach einem Versand nicht geleert wurde.

Literaturverzeichnis

[1] Offizielle MPI-Dokumenation: http://www.mpi-forum.org/docs/, Zugriff 22.10.2012

[2] Tutorial zur Benutzung der Software Git: http://www.githowto.com, Zugriff 22.10.2012

[3] MPI-Hilfe der TU Chemnitz: http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/isend.html, Zugriff 22.10.2012

[4] Interessanter Artikel zur Speicherverwaltung in C http://de.wikibooks.org/wiki/C-Programmierung:_Speicherverwaltung, Zugriff 22.10.2012