Ausarbeitung zum Transputer-Praktikum
Sonntag, 12. Juni 1995

Kay Römer
Thomas Holubar
Frank Pilhofer
Atilla Ünlü

Game of Life

Aufgabe 2

Aufgabenstellung

Das Game of Life simuliert Leben und Sterben in einer virtuellen Welt unter einfachsten Bedingungen. In einer zweidimensionalen Matrix sind Zellen angeordnet, für die es nur die Zustände "Leben" und "nicht Leben" sowie nur eine einzige Lebensbedingung gibt: die Anzahl der Nachbarn. Hat eine Zelle die "richtige" Anzahl Nachbarn, so überlebt sie bzw. wird eine neue Zelle geboren, bei der "falschen" Anzahl Nachbarn stirbt die Zelle. Dabei lassen sich die Parameter, die die Dynamik des Systems bestimmen, frei wählen.

Technisch jedenfalls beschränkt sich das Problem auf das Zählen von Nachbarn.

Anforderungen

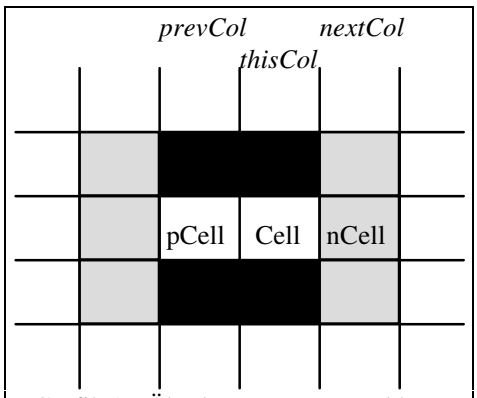
Folgende Überlegungen bzw. Leistungsanforderungen soll das Programm erfüllen:

- Die Ausgabe der Lebensmatrix soll als ASCII-Grafik bzw. in einem X11-Fenster erfolgen, oder auch zur besseren Zeitmessung abschaltbar sein.
- Um bei der X11-Ausgabe zeitraubende Umwandlungen zu vermeiden, wird die Matrix innerhalb des Programms als X11-Bitmap behandelt, d.h. horizontal werden 8 Zellen in einem Byte gespeichert, und die Zeilenanfänge sind auf Bytegrenze ausgerichtet. Diese Technik erweist sich als vorteilhaft in Bezug auf Speicherausnutzung und wirkt sich in Hinsicht auf die zu übertragende Datenmenge positiv aus.
- Die logischen Kommunikationskanäle sollen so gewählt werden, daß Daten stets ohne Umwege direkt verschickt werden können, d.h. daß Daten von *A* nach *B* niemals Drittprozesse passieren müssen.
- Der initiale Zustand der Matrix soll entweder aus einer Datei geladen (wobei sich natürlich das X11-Bitmap Dateiformat anbietet) oder zufällig gewählt werden können.

Sequentielle Optimierung

Bevor die Parallelisierung des Problems angegangen wird, sollte betrachtet werden, wie die Lösung des sequentiellen Problems beschleunigt werden kann. Denn Grundlage einer schnellen parallelen Implementierung ist meist eine schnelle sequentielle Implementierung. Wie bereits erwähnt ist das einzige technische Problem die Zählung der Nachbarn einer Zelle.

Gegenüber dem naiven Ansatz, zu jeder einzelnen Zelle alle acht Nachbarzellen abzuklappern und das Leben in ihnen zu zählen, ergibt sich ein hohes Optimierungspotential. So erkennt man (Grafik 1), daß bei zwei benachbarten Zellen vier Nachbarn (schwarz) identisch sind. Geht man von links nach rechts vor, so ist bei der Bestimmung der Nachbarn von *Cell* auch der Zustand von *pCell* bekannt, d.h. es müssen nur die Zustände der drei rechten Nachbarn neu abgefragt werden.



Grafik 1 - Überlappung von Nachbarn

Die Grafik enthält bereits die im Programm verwendete Nomenklatur:

Cell	die Zelle selber
nCell	die Zelle rechts von Cell
prevCol	Anzahl der Nachbarn von *Cell* in der linken Spalte
thisCol	Anzahl der Nachbarn von *Cell* in dieser Spalte, d.h. direkt über und unter *Cell*; die Zelle selbst gehört ja nicht zu ihrer eigenen Nachbarschaft
nextCol	Anzahl der Nachbarn von *Cell* in der rechts folgenden Spalte, exklusive der Zelle in der gleichen Zeile (*nCell*)

Mit dieser Bezeichnung berechnet sich die Anzahl der Nachbarn der Zelle *Cell* einfach als

$$Nachbarn = prevCol + thisCol + nextCol + nCell$$

Bei der einmaligen Berechnung der Nachbarn ist bislang noch nichts gewonnen. Gespart wird erst, wenn die Berechnung weiter nach rechts fortgesetzt wird. Denn die Werte der meisten obigen Parameter im Folgezustand sind bereits bekannt:

$$\begin{aligned} Cell_{i+1} &= nCell_i \\ prevCol_{i+1} &= thisCol_i + Cell_i \\ thisCol_{i+1} &= nextCol_i \end{aligned}$$

Lediglich die Werte von $nextCol_{i+1}$ und $nCell_{i+1}$ müssen neu bestimmt werden, was genau drei Zugriffen auf die Speichermatrix entspricht, und damit gegenüber 8 Zugriffen ein deutlicher Gewinn ist.

Doch sogar diese drei Zugriffe lassen sich anhand der Kenntnis über den Aufbau der Bitmatrix weiter optimieren. Ein Zugriff auf eine Zelle erfolgt, indem ein Speicherwert geladen und mit einer bestimmten Bitmaske verknüpft wird, um das einzelne Bit als Zustand der Zelle zu isolieren. Ein naiver Algorithmus würde für jeden Zugriff ① die Speicheradresse berechnen und ② die Bitmaske neu berechnen, was einem schrecklichen

Haufen arithmetischer Operationen inklusive Multiplikationen entspricht, z.B. mit folgenden Makros zum Setzen, Löschen und Abfragen eines Bits an der Position (x,y) in der Matrix f:

```
#define BsSetCell(f,x,y)           \
     (*((f) + (y) * thesize.lofl + ((x)>>3)) |= (0x01 << ((x) & 0x07)))
#define BsResetCell(f,x,y)         \
     (*((f) + (y) * thesize.lofl + ((x)>>3)) &= ~(0x01 << ((x) & 0x07)))
#define BsGetCell(f,x,y)           \
     (!!(*((f) + (y)*thesize.lofl + ((x)>>3)) & (0x01 << ((x) & 0x07))))
```

Wiederum läßt sich das Verfahren für den Fall, daß horizontal von links nach rechts vorgegangen wird, optimieren. Betrachten wir den Zugriff auf die Zellen *Cell* und *nCell* der Grafik 1.

- Innerhalb eines Bytes ist die Bitmaske von *nCell* die um eine Position nach links geshiftete Bitmaske von *Cell*. Falls sich nCell auf einer Bytegrenze befindet, so müßte die Bitmaske neu auf 0x01 initialisiert werden.
- Da in der Bitmap jede Zeile auf einer Bytegrenze beginnt, sind die Bitmasken innerhalb einer Spalte (für die Zellen oberhalb und unterhalb von *nCell*) identisch, nur die Speicherposition ist verschieden.

Das Verfahren wird daher wie folgt optimiert: Es werden drei Pointer für die vorige, die jetzige und die folgende Zeile sowie die Bitmaske initialisiert. Beim Übergang von einer auf die folgende Spalte wird die Bitmaske um eine Position nach links geshiftet. Wird sie dabei zu 0 (das eine Bit ist nach links herausgefallen), so werden die drei Pointer inkrementiert und die Bitmaske neu auf 0x01 zurückgesetzt.

Bei jeder Berechnung von *nextCol* und *nCell* werden gegenüber obigem `BSGetCell` Makro drei Multiplikationen, sechs Additionen und fünf (6-1) Bitshifts eingespart. In einer (x,y)-Matrix, in der diese Werte über den Daumen gepeilt auch x*y mal benötigt werden, zahlt sich die Optimierung in erheblich verbesserter Laufzeit aus.

Parallelisierung

Grundüberlegungen

Die Berechnung wird parallelisiert, in dem jedem Prozess ein Teil der Gesamtmatrix zur Berechnung übergeben wird. Die horizontale Orientierung der Bits erschwert eine vertikale Aufteilung, und würde auch die optimierten Algorithmen der Nachbarzählung beeinträchtigen. Deshalb wird unser digitaler Lebensraum in horizontale Streifen aufgeteilt. Stehen beispielsweise 3 Prozesse zur Verfügung, und ist eine Matrix von 100 Zeilen zu berechnen, so erhält der erste Prozess die Zeilen 0-33, der zweite 34-66 und der dritte die Zeilen 67-99.

Wiederum wird nach dem Master-Slave-Prinzip vorgegangen, d.h. die Verteilung und Steuerung der Aufgaben findet in einem spezialisierten Master statt, während alle Berechnungen von möglichst dummen, identischen Slaves ausgeführt werden.

Zusatzzeilen

Es fällt schnell auf, daß die beschriebene disjunkte Zerlegung der Matrix nicht ausreicht, denn damit Prozeß 1 die Nachfolgezustände in Zeile 33 berechnen kann, muß ihm der Zustand der Zeile 34 bekannt sein. Ebenfalls benötigt Prozeß 2 die Zeile 33, um seine neue Zeile 34 berechnen zu können.

Zwar könnte der Master bei der Übertragung der Matrix an die Slaves jeweils eine überlappende Zeile verschicken, doch auch dies verschiebt das Problem nur um einen Berechnungsschritt. Denn nach dem ersten Schritt hat sich die Zeile 34 in Prozeß 2 *verändert*, und diese Veränderung muß wiederum an Prozeß 1 übertragen werden bevor der zweite Berechnungsschritt beginnen kann.

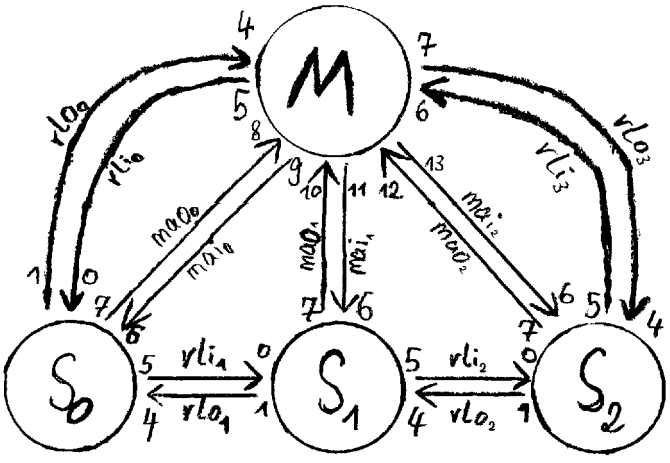
Nach jedem einzelnen Berechnungsschritt muß deshalb die letzte Zeile eines jeden Prozesses an den Nachfolger übermittelt werden, und die erste Zeile eines jeden Prozesses an den Vorgänger.

Kommunikationsstruktur

Bei den Leistungsanforderungen wurde aufgezählt, daß die Kommunikationsstruktur *ideal* gewählt werden soll, damit jede Art von Datenübertragung direkt erfolgen kann. Für den eben genannten Datenaustausch der überlappenden Zeilen, bei dem jeder Prozess mit seinem Vorgänger und Nachfolger kommunizieren muß, bietet sich eine bidirektionale Ringstruktur an.

Weiterhin ist nach jedem Berechnungsschritt die Teilmatrix eines jeden Slaves an den Master zu übertragen. Gerade diese Datenmenge sollte auch einen kürzstmöglichen Weg laufen, wofür eine unidirektionale Verbindung zum Master benötigt wird. Da Prozesse andererseits auch individuelle Befehle des Masters erhalten können, wie z.B. die Festlegung der Größe einer Teilmatrix, wird die direkte Slave-Master-Kopplung ebenfalls bidirektional ausgelegt.

Unsere gewünschte Kommunikationsstruktur sieht mit dem Master M und den drei Slaves S_0, S_1 und S_2 wie folgt aus:



Grafik 2 - Kommunikationsstruktur

CDL-Manie

Grafik 2 zeigt die Kommunikationsstruktur, wie wir sie uns wünschen. Doch - läßt sie sich auch realisieren? Sie läßt sich leider nicht wie gewohnt in einem kompakten CDL-Einzeiler dynamisch definieren. Zwar sind die beiden einzelnen Elemente, ein bidirektionaler Ring und eine bidirektionale Farm, kein Problem, doch ließen sie sich nicht zusammenführen. Deshalb wird der Weg gewählt, jedem Prozess mittels einer Komponentendeklaration explizit benannte Kommunikationskanäle zuzuweisen. Das Verfahren hat leider den Nachteil, daß es nicht flexibel ist; für jede gewünschte Anzahl von Slaves müssen neue Kanäle definiert werden. Um dennoch die Konfiguration möglichst flexibel zu halten, wurde das Programm `cdlc.c` geschrieben, welches zu einer gewünschten Anzahl von Prozessen eine fertig kompilierbare CDL-Datei erzeugt. Das Ergebnis für obiges Beispiel mit 3 Slaves sieht folgendermassen aus:

```
component A[i] { code master.z;
             puid /Cluster/00;
                   streams  , , , ,
                          <| rlo{0},   >| rli{0},
                          <| rli{i},   >| rlo{i},
                          <| mao{0},   >| mai{0},
                          <| mao{1},   >| mai{1},
                          <| mao{2},   >| mai{2}
             ;
            }
component B[i] { code slave.z;
                   streams <| rli{i},   >| rlo{i}, , ,
                          <| rlo{i+1}, >| rli{i+1},
                          <| mai{i},   >| mao{i}
             ;
            }
A{$1} [i < $1] ^^ B{i}
```

Komponente *A* repräsentiert den Master, die replizierten Komponenten *B* die Slaves. In der Taskforce werden keine Kommunikationsstrukturen definiert; alle Kanäle sind benannt und in den Komponentendeklarationen angegeben. Die Slaves sind von 0 bis n-1 (mit n=3) numeriert. In Grafik 2 sind bereits die Namen und Filedeskriptoren der Kanäle eingetragen.

Übrigens ist die Komponentendeklaration *B* der Slaves unabhängig von deren Anzahl. Genaugenommen tut der CDL-Generator nichts weiter als die entsprechende Anzahl Zeilen mit *mai{i}* und *mao{i}* Kanälen zur direkten Kommunikation des Masters mit dem jeweiligen Slave zur Definition der Komponente *A* hinzuzufügen.

Beim Anblick der Grafik fällt auch auf, daß die Kanäle vom Master zum ersten bzw. zum letzten Slave doppelt vorhanden sind. Dank der zusätzlichen Kanäle sehen die Filedeskriptoren für alle Slaves gleich aus; ansonsten müßten für den ersten und den letzten Slave Ausnahmen gemacht werden. Da in dieser Implementation die dummen Slaves noch nicht einmal wissen, ob sie die ersten oder letzten sind, werden die vier Kanäle gerne geopfert.

Kommunikation

Dieses Kapitel beschreibt, welche Daten auf welchen Wegen im Lauf des Programms durch die Gegend gefrachtet werden.

Befehle

Es gibt drei Befehle, die den globalen Zusammenhang des Systems steuern. Diese Befehle laufen vom Master über den Ring zurück zum Master, damit alle Prozesse den Befehl erhalten. Befehle werden in der folgenden Struktur verschickt:

```
typedef struct {
  int command;             /* Typ des Kommandos                */
  int prozessor;           /* Prozessoren-Zähler               */

  int steps;               /* Anzahl der Berechnungsschritte   */
  int SurviveMin;          /* Überlebensregeln des Game of Life */
  int SurviveMax;
  int BirthMin;
  int BirthMax;
} comm;
```

`command` kann die folgenden drei Werte annehmen:

COUNT Zählen der Prozesse im System. Vor dem Abschicken des Befehls links in den Ring hinein wird das Feld `prozessor` vom Master auf 0 initialisiert. Jeder Slave merkt sich das Feld als eigene Identifikation (auch wenn diese nirgends verwendet wird) und erhöht das Feld um 1. Wenn der Master das Paket am rechten Ende des Rings empfängt, enthält `prozessor` die Anzahl der Prozesse im System. Dieser Befehl wird nicht nur am Anfang verschickt; er wird gleichzeitig zur Re-Initialisierung verwendet, d.h. nach diesem Befehl erwartet ein Slave eine neue Bitmap.

GO Startet die Berechnung in den Slaves. `steps` enthält die Anzahl von durchzuführenden Schritten, und `SurviveMin` etc. enthalten die Spielregeln, anhand derer das Game of Life durchgeführt werden soll. Jeder Slave muß das Paket natürlich weiterschicken, bevor er die Berechnung startet.

QUIT Nach Erhalt und Weiterleitung eines Pakets mit diesem Befehl beendet sich der Slave.

Nach Abarbeitung eines **GO**-Befehls kann ein Slave jeden der drei Befehle empfangen. Ein weiterer **GO**-Befehl führt die Berechnung weiter, **QUIT** beendet den Slave, und **COUNT** stößt eine Re-Initialisierung an, in deren Verlauf der Slave auch eine neue Bitmap vom Master erhält. Dieses Prinzip enthält schon die gewünschte Optimierung, daß die Bitmap nur genau einmal vom Master zum Slave übertragen wird.

Bitmaps

Am Anfang einer Berechnung muß eine initiale Bitmap vom Master zum Slave übertragen werden, und nach jedem Berechnungsschritt wird das Ergebnis zum Zweck der Ausgabe vom Slave an den Master übertragen. Empfangen wird eine Bitmap im Slave immer nach einem **COUNT**-Befehl. Zu diesem Zeitpunkt weiß der

Slave noch nicht, wie groß die Bitmap sein wird; deshalb schickt der Master an dieser Stelle zur Information ein Paket folgender Struktur voraus:

```
typedef struct {
  int xsize;
  int ysize;
  int yoffs;
  int length;
  int lofl;
} bmsize;
```

Dabei enthalten `xsize` und `ysize` die Größe der Teilmatrix des jeweiligen Prozesses. `yoffs` enthält die vertikale Anfangsposition der Teilmatrix. In dem bereits erwähnten Beispiel mit 100 Zeilen und 3 Prozessen ist für den ersten Prozess `yoffs=0`, für den zweiten `yoffs=34` und für den dritten `yoffs=67`. Dieser Wert ist für den Slave völlig uninteressant, doch wird ein identisches Paket für jeden Slave im Master gespeichert, der den Wert zur Positionierung der Teilmatrix auf dem Bildschirm braucht.

`length` ist die Länge der Bitmap für diesen Prozess in Bytes, d.h. die Anzahl an Bytes, die jeweils übertragen werden muß. `lofl` ist ein verwandter Wert für die Länge einer einzelnen Zeile in Bytes, benötigt beim Ringupdate. Beide Werte sind im Prinzip redundant, denn `lofl=(xsize+7)/8`, und `length=lofl*ysize`. Doch da die Übertragung von 12 halbwegs überflüssigen Bytes zeitlich nahezu kostenlos ist, wurde hier auf Bytediät verzichtet.

Nach Übertragung dieses Paketes allokiert der Slave genügend Speicherplatz und empfängt die Matrix.

Bei der Rückübertragung nach jedem Berechnungsschritt wird auf das Senden des "Informationspaketes" verzichtet, da sich der Master die Größe der Bitmap jedes einzelnen Slaves gemerkt hat.

Übrigens werden bei der Übertragung der Bitmaps an die Slaves (oder zurück) keine überlappenden Zeilen verschickt, dies wird durch ein direkt auf die Übertragung folgendes Ringupdate viel bequemer erschlagen.

Zum Schluß noch ein Wert: Bei dem Beispiel einer Matrix der Kantenlänge 100 Pixel, die auf 3 Slaves aufgeteilt wird, erhält jeder Slave eine Teilmatrix mit 33 (bzw. 34) Zeilen; jede zu verschickende Teilbitmap enthält 3432 (bzw. 3536) Bits beziehungsweise 429 (bzw. 442) Bytes. Die anfallende Datenmenge hält sich also in Grenzen.

Das Ringupdate

Wie bereits erwähnt, benötigt jeder Slave im System zur Berechnung seiner Teilmatrix zwei zusätzliche Matrixzeilen: die letzte Zeile des Vorgängers und die erste Zeile des Nachfolgers, und muß gleichzeitig seine eigene erste und letzte Zeile exportieren. Die beiden importierten Zeilen müssen nach jedem Berechnungsschritt neu aufgefrischt werden. Daten müssen nur zum Vorgänger bzw. Nachfolger übertragen werden, weshalb die Kommunikation sinnvollerweise über den bidirektionalen Ring abgewickelt. Der komplette Prozeß wurde schließlich *Ringupdate* getauft.

Gerade bei dem Ringupdate hat es sich gezeigt, daß neben guten Algorithmus auch ein paar gute Überlegungen die Laufgeschwindigkeit und Effizienz deutlich verbessern können. Doch betrachten wir zunächst eine naheliegende Möglichkeit des Ringupdates.

Auch wenn dies noch nicht die optimale Version des Ringupdates ist, soll sie hier genau beschrieben werden. Einerseits, weil der Text schon geschrieben war als sich die Verbesserungen ergaben, und andererseits, damit der Text später auch ohne allzu viel Text noch verständlich ist.

Das Ringupdate wird abgewickelt, indem zunächst die erste Zeile eines jeden Prozesses in rechter Richtung[1] durch den Ring geschickt wird, und anschließend läuft die letzte Zeile eines jeden Prozesses in linker Richtung zurück. Bildlich dargestellt spielt der Master Tischtennis mit sich selbst: Der Ball in Form einer Matrixzeile wird in den Ring geschmettert, durchfliegt alle Slaves, und bei der Wiederankunft im Master am rechten Ende des Rings wird das Datenpaket retourniert. Stören an diesem Bild tut nur, daß sich der Ball auf jedem Punkt seiner Reise ändert, schließlich schickt jeder Slave eine andere erste/letzte Zeile weiter.

Für den Slave genügen zur Abwicklung des so spezifizierten Ringupdates vier Schritte:

① von "links" (Kanal rli_n) wird die letzte Zeile des vorherigen Slaves gelesen
② die letzte Zeile wird nach "rechts" (Kanal rli_{n+1}) herausgeschrieben
③ von "rechts" (Kanal rlo_{n+1}) wird die erste Zeile des Nachfolgers gelesen
④ die erste Zeile wird nach "links" (Kanal rlo_n) herausgeschrieben.

Global gesehen sieht das Ringupdate wie folgt aus:

① Der Master schreibt eine Leerzeile "links" (Kanal rli_0) in den Ring hinein, wo diese vom ersten Slave als letzte Zeile des Vorgängers gelesen wird.
② Wie oben beschrieben schreibt der Slave seine letzte Zeile heraus, welche sofort vom Nachfolger gelesen wird. Dieser Punkt setzt sich bis zum letzten Slave fort.
③ Der Master liest am "rechten" Ende des Rings (Kanal rli_n) die letzte Zeile des letzten Slaves (und vergißt sie - sie wird nicht gebraucht). Alle Slaves warten laut obigem "Regelwerk" jetzt vor Punkt 3.
④ Der Master schreibt eine Leerzeile nach rechts in den Ring hinein (Kanal rlo_n), wo sie vom letzten Slave als erste Zeile des Nachfolgers gelesen wird.
⑤ Analog zu Punkt ② schreibt der Slave seine erste Zeile heraus, welche sofort vom Vorgänger gelesen wird.
⑥ Der Master liest am "linken" Ende des Rings (Kanal rlo_0) die erste Zeile des ersten Slaves (und vergißt sie). Das Ringupdate ist beendet.

Das ganze hört sich wahrscheinlich noch komplizierter an als es ist. In Programmzeilen gegossen sieht der Prozeß wie folgt aus:

---

[1] Richtungsangaben im Ring sind am besten anhand von Grafik 2 zu verstehen: Gemeint ist die Richtung, die die Nachricht zwischen den Slaves läuft. "links" bedeutet in Richtung absteigender Slavenumerierung und von 0 zum Master; "rechts" bedeutet in Richtung aufsteigender Slavenumerierung.

Ringupdate (Slave)

```
RECV (RING_LIN,                          /* letzte Zeile des Vorgängers empfangen */
      BFieldCurrentAddress,              /* Pointer auf 0. Zeile der Teilmatrix   */
      thesize.lofl,                      /* Länge einer Zeile in Bytes            */
      return 1);
SEND (RING_ROUT,                         /* eigene letzte Zeile nach rechts raus  */
      BFieldCurrentAddress+(thesize.ysize*thesize.lofl),
      thesize.lofl,
      return 1);
RECV (RING_RIN,                          /* erste Zeile des Nachfolgers empfangen */
      BFieldCurrentAddress + (thesize.ysize + 1) * thesize.lofl,
      thesize.lofl,
      return 1);
SEND (RING_LOUT,                         /* eigene erste Zeile nach links raus    */
      BFieldCurrentAddress + thesize.lofl,
      thesize.lofl,
      return 1);
```

Ringupdate (Master)

```
SEND (RING_LI0, empty_line, lofl, exit (1));  /* leere Zeile nach links raus      */
RECV (RING_LII, trash,      lofl, exit (1));  /* rechts zeile lesen und vergessen */
SEND (RING_LOI, empty_line, lofl, exit (1));  /* leere Zeile nach rechts raus     */
RECV (RING_LO0, trash,      lofl, exit (1));  /* links Zeile lesen und vergessen  */
```

Die Makros SEND und RECV machen übrigens nichts weiter als die entsprechende Funktion in PsxIO aufzurufen und zu testen, ob ein Fehler aufgetreten ist; im Fehlerfall wird eine Fehlermeldung ausgegeben und ein Befehl ausgeführt.

Protokoll einer Berechnung

Die einzelnen Elemente der Kommunikation sind jetzt beschrieben. Dieses Kapitel erläutert nun deren zeitliche Abfolge.

① Der Master verschickt über den Ring einen **COUNT**-Befehl und zählt die Prozesse.

② Der Master bestimmt die Größe der Gesamtbitmap, teilt die Zeilen durch die Anzahl an Prozessen und verschickt die jeweilige Bitmapgröße über die direkten Kanäle an die einzelnen Slaves.

③ Die initialen Teilbitmaps werden über die direkten Kanäle verschickt.

④ Ringupdate

⑤ Über den Ring wird ein **GO**-Befehl verschickt, in dem die Anzahl der durchzuführenden Schritte und die Regeln des Game of Life eingetragen sind. In den Slaves startet die Berechnung der Folgegeneration.

⑥ Ringupdate

⑦ Der Master liest über die direkten Kanäle die Bitmaps der Folgegeneration aus und stellt sie auf dem Bildschirm dar. In ⑥ wurde bereits das Update durchgeführt, deshalb kann jeder Slave sofort nach Auslesen der Bitmap mit dem nächsten Berechnungsschritt starten. Mit anderen Worten, während der Master noch von den hinteren Prozessen die Bitmaps ausliest, sind die ersten Prozesse schon wieder munter am Rechnen.

⑧ Solange noch nicht die gewünschte Anzahl von Berechnungsschritten (wie im GO-Befehl angegeben) erfolgt ist, gehe zu ⑥.

Falls der Benutzer weitere Generationen zu sehen wünscht, mache weiter bei ⑤. Falls eine neue Bitmap berechnet werden soll, gehe nach ①. Falls der Benutzer das Programm beendet, mache weiter.

⑩ Sende **QUIT**-Befehl durch den Ring. Die Slaves beenden sich. Der Master beendet sich ebenfalls, nachdem er das **QUIT**-Paket am rechten Ende des Rings empfangen hat.

Optimierung des Ringupdates

Vorhin wurde eine Prozedur zum Ringupdate angegeben, die zwar funktioniert, doch noch verbesserbar ist. Daß eine Verbesserung not tut, fällt im Programmtext nicht auf; alle Bits werden gut gepackt und nur ein einziges Mal über die Leitung geschickt. Klarer wird das Problem in einem Zeitdiagramm, das hier für die oben aufgezählten Schritte ⑥ (Ringupdate) und ⑦ (Ergebnisbitmap versenden) aufgestellt werden soll. Die folgende Tabelle stellt auf der Zeitachse das Senden und Empfangen von Paketen in Master und drei Slaves während eines Ringupdates dar. Die Tabelle ist wie der Ring zirkulär zu sehen; am rechten Ende steht wieder der Master. Jede Spalte (jeder Prozess) ist weiter in einen linken und rechten Teil unterteilt; dies entspricht der linksgerichteten bzw. rechtsgerichteten Kommunikation im Ring. Jeder Pfeil entspricht einem versendeten bzw. empfangenen Paket (in jeder Zeiteinheit gibt es einen Sender und Empfänger). Waagerechte Pfeile verlaufen im Ring, vertikale auf den direkten Kanälen zum Master. Als Zeiteinheit wird die Paketanzahl genommen, was laut Aufgabe 1b berechtigt ist (Pakete verschiedener Größe werden in ungefähr gleicher Zeit verschickt).

Zeit	Master	S_0	S_1	S_2
1	→	→		
2		→	→	
3			→	→
4	→			→
5	←			←
6			←	←
7		←	←	
8	←	←		
9	↑	↑		
10	↑		↑	
11	↑			↑

Die Zeitverschwendung ist offensichtlich. Das Ringupdate vollzieht sich wie beim Ping Pong und dauert von Schritt 1 bis Schritt 8. Erst in Schritt 9 kann der Master mit dem Einsammeln der Teilergebnisse beginnen. Und erst nachdem die Bitmap fertig abgeholt wurde, kann ein Slave mit seiner Berechnung fortfahren. Im ersten Slave werden so 4 Zeiteinheiten durch Warten vergeudet, im zweiten 5 und im letzten sogar 6. Bei mehr als drei Prozessoren steigt die Wartezeit sogar weiter an. Als Übeltäter wird natürlich das Ping-Pong-Spiel während des Ringupdates entlarvt.

Es zeigt sich, daß das hin und her tatsächlich durch eine einfach Änderung des Updateprotokolls vermieden werden kann. In der folgenden Version "antwortet" jeder Slave *sofort* seinem linken Nachbarn mit der Ausgabe

der eigenen ersten Zeile, bevor die letzte Zeile weitergeschickt wird. Das mag sich verworren anhören; im Zeitdiagramm sieht das veränderte Ringupdate folgendermassen aus:

Zeit	Master	S_0	S_1	S_2
1	→	→		
2	←	←		
3		→	→	
4		←	←	
5			→	→
6			←	←
7	→			→
8	←			←
9	↑	↑		
10	↑		↑	
11	↑			↑

Laut dem neuen Diagramm ist noch exakt überhaupt nichts gewonnen, doch immerhin wurde die Übertragung aufgeräumt. In jedem Slave ist nur noch eine Unterbrechung zum Ringupdate nötig, anstatt der zwei Unterbrechungen zum Hin- und Herschaufeln der ersten Fassung. Störend ist nun noch die Tatsache, daß jeder Slave warten muß, bis der Master das Ringupdate beendet hat und die Bitmaps ausliest.

Dieser Zustand läßt sich verbessern, indem die starre Sequentialisierung von Ringupdate und Übertragung der Bitmaps aufgegeben wird. Wird die Übertragung der Bitmap in die Mitte des Ringupdates eines jeden Slaves gestellt, ergibt sich folgende Tabelle.

Zeit	Master	S_0	S_1	S_2
1	→	→		
2	←	←		
3	↑	↑		
4		→	→	
5		←	←	
6	↑		↑	
7			→	→
8			←	←
9	↑			↑
10	→			→
11	←			←

Zwar werden insgesamt immer noch 11 Zeitschritte benötigt, doch zeigt sich hier eine echte Verbesserung. Denn Slave 0 ist hier bereits in Zeitschritt 5 mit seinem Update fertig und kann in Schritt 6 mit der Berechnung seiner nächsten Generation beginnen, was im alten Verfahren erst in Schritt 10 möglich war. Slave 1 beginnt in 9 anstatt 11, doch da das Update und die Berechnung in einer Schleife durchgeführt werden, kann Slave 1 noch drei Schritte weiterrechnen, während sein Vorgänger, Slave 0, bereits wieder ins Update verwickelt ist.

Das mag sich bei drei Slaves noch nicht allzu dramatisch anhören, doch bei jedem weiteren Prozessor werden zwei zusätzliche Zeiteinheiten gewonnen. Um einen Wert zu nennen: bei dem späteren Beispiel einer Bitmap von 640x400 auf 16 Prozessoren ergibt sich eine Effizienzsteigerung von 84.2% auf 93.4%, d.h. die

Menge der zu 100% fehlenden Effizienz wurde mehr als halbiert. Dafür, daß nur einige wenige Programmzeilen *verschoben* worden sind, ist diese Leistungssteigerung ziemlich beeindruckend.

Die Benutzerschnittstelle

Nicht vergessen werden sollte ein ebenfalls wichtiger Aspekt eines jeden interaktiven Programms. Die Benutzerschnittstelle von Life ähnelt der eines Kommandointerpreters: nach dem Programmstart erscheint der Prompt `mp-life>` und das Programm wartet darauf, daß der Benutzer ein Kommando eingibt. Kommandos bestehen aus einem Namen, der optional von Parametern gefolgt wird.

Verfügbare Kommandos

In der Beschreibung der Kommandos werden Platzhalter, die durch tatsächliche Werte ersetzt werden müssen, kursiv gedruckt. Optionale Teile werden in eckige Klammern eingeschlossen ('[...]'), Alternativen werden in geschweifte Klammern eingeschlossen und durch Pipezeichen voneinander getrennt ('{Alternative1|Alternative2|...}'). Hier nun eine Erläuterung aller verfügbarer Kommandos:

`help`

Das wichtigste Kommando zuerst, es zeigt eine Übersicht über alle verfügbaren Kommandos an.

`exit`

Beendet Life.

`read` ***`dateiname`***

Dieses Kommando liest eine Bitmap ein, die als "Grundbild" für Life verwendet wird. *`Dateiname`* sollte der Name einer Bitmapdatei im XBM[2] Format sein. Solche Dateien haben oft die Erweiterung .xbm; diese muß mit angegeben werden! Eine schier endlose Zahl solcher Bitmaps finden sich im Verzeichnis /usr/include/X11/bitmaps. Alternativ kann man sich auch welche mit dem Programm **bitmap** selbst zeichnen, oder mit dem Programm **xv** bestehende Grafiken in das Format wandeln.

`random` ***`Breite Höhe`*** **`{`*****`Belegung`*** **`| - }`**

`Random` erzeugt ebenfalls ein "Grundbild" für Life mit der Breite *`Breite`* und der Höhe *`Hoehe`*. *`Belegung`* ist ein Maß dafür, welcher Anteil der Pixel gesetzt sein soll (in Prozent, zulässig sind Werte von 0 bis 100). Gibt man statt dessen '−' an, so werden etwa die Hälfte aller Pixel gesetzt.

`seed {`***`Wert`*****`|-}`**

Setzt den Anfangswert des für `random` verwendeten Zufallsgenerators. Dabei kann für *`Wert`* eine beliebige Ganzzahl angegeben werden. Gibt man statt dessen '−' ein, so wird der hoffentlich einigermaßen zufällige aktuelle Wert der Systemuhr verwendet. Nach Aufruf dieses Kommandos mit einem konstanten Wert liefert `random` stets die gleiche Bitmap, was zu Testzwecken sehr nützlich ist.

---

[2] **X** Window **B**i**tm**ap

scale *X-Skalierung Y-Skalierung*

Mit diesem Kommando kann die Vergrößerung der Graphikausgabe über RX11 eingestellt werden. *X-Skalierung* stellt die Vergrößerung in horizontaler, *Y-Skalierung* die Vergrößerung in vertikaler Richtung ein. Beide Werte müssen zwischen 1 und 8 liegen.

display {text|x11|off}

Stellt den Anzeigemodus ein. Bei "text" erfolgen die Ausgaben auf dem Terminal, bei "off" werden (zur Messung der Berechnungsgeschwindigkeit) die Ausgaben ganz abgeschaltet. Der Parameter "x11" schließlich veranlaßt Life dazu, die Ausgaben in einem X11 Fenster unter Benutzung von Remote X11 (RX11) vorzunehmen.

delay *Verzoegerung*

Dieses Kommando setzt die zeitliche Verzögerung zwischen den Ausgaben der einzelnen Iterationsschritte. Mit *Verzoegerung* wird die Wartezeit in Millisekunden angegeben. Ein Wert größer Null erweist sich allerdings bei größeren Bitmaps als unnötig, da die Berechnungen für genügend Verzögerung sorgen.

rules *BirthMin BirtMax SurviveMin SurviveMax*

Setzt die Parameter für den Life Algorithmus. Ist eine Zelle im aktuellen Iterationsschritt tot und sind ≥ *BirthMin* und ≤ *BirthMax* Nachbarzellen lebendig, so wird die Zelle "geboren", d.h. sie wird im nächsten Iterationsschritt lebendig sein. Ist eine Zelle im aktuellen Iterationsschritt lebendig und sind ≥ *SurviveMin* und ≤ *SurviveMax* Nachbarzellen lebendig, so wird die Zelle lebendig bleiben ("überleben"), andernfalls wird sie für tot erklärt. Bevor dieser Befehl aufgerufen wird, bleiben die Standardwerte BirthMin=3, BirthMax=3, SurviveMin=2 und SurviveMax=3 gesetzt.

run *Schritte*

Startet die Simulation für die angegebene Anzahl von Schritten unter Beachtung der aktuellen Werte für Verzögerung, Anzeigemodus, Vergrößerung und Regeln. Als "Ausgangsbild" wird das Ergebnis der letzten Simulation verwendet, es sei denn es wurde in der Zwischenzeit ein read oder random Kommando abgesetzt. In diesem Falle wird das neu geladene bzw. erzeugte Ausgangsbild verwendet.

Graphikausgabe

Zur Graphikausgabe wurde die Remote X11 (RX11) Bibliothek verwendet. Die Funktionen dieser Bibliothek sind ausführlich in der Dokumentation zu Remote X11 beschrieben. Hier soll nur darauf eingegangen werden, wie die Bibliothek zur Realisierung der Graphikausgaben für Life verwendet wurde.

Um aufwendige Konvertierungen der von den Slaves gelieferten Daten zu vermeiden, werden alle Berechnungen direkt auf Bitmaps im X11 Format durchgeführt. Dieses weist folgende Merkmale auf:

- In einem Byte sind 8 Pixel untergebracht.
- Jede Zeile beginnt mit einem neuen Byte.

- Die Bitorder in einem Byte ist Little Endian, d.h. das niederwertigste Bit entspricht dem am weitesten links stehenden, das höchstwertigste dem am weitesten rechts stehenden Pixels in einem Block von acht Pixeln.

Die Aufgabe des Life Serverprogramms besteht nun noch darin, eine derartige Bitmap aus einer XBM Datei zu erzeugen, die Bitmap nach Wunsch zu vergrößern und schließlich in das Ausgabefenster zu kopieren. Diese Aufgaben sind recht einfach zu bewerkstelligen, da entsprechende Funktionen in der RX11 Bibliothek vorhanden sind.

`RXReadXbmFile()` liest eine XBM Datei und liefert die darin enthaltenen Daten im X11 Bitmap Format. Mit `RXScaleBitmapData()` können dann derartige Bitmapdaten vergrößert und mittels `RXCopyBitmap()` in das Ausgabefenster kopiert werden.

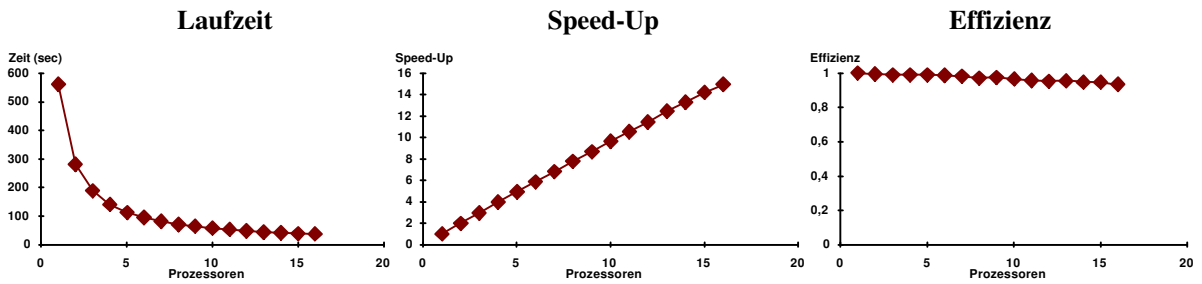
Performance

Zur Messung der Performance der parallelen Implementation wurden stets 100 Generationen des gleichen Zufallsfelds der Größe 640x400 mit abgeschalteter Bildschirmausgabe berechnet, wozu das folgende kurze Skript verwendet wurde (siehe Beschreibung der Benutzerschnittstelle).

```
display off
seed 42
random 640 400 -
run 100
exit
```

Trotz der abgeschalteten Ausgabe wurden die Ergebnisbitmaps immer noch brav nach jedem Einzelschritt an den Master übertragen, so daß die Ergebnisse nicht allzu realitätsfern sind. Alle Messungen fanden auf der Architektur 4x4 statt. Der Master wird auf dem Rootprozessor plaziert, die Anordnung der Slaves wurde vollständig Helios überlassen.

Prozessoren	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Zeit (s)	560.9	281.2	188.6	141.4	113.5	95.2	82.4	71.8	64.4	58.0	53.2	49.0	45.0	42.2	39.5	37.5
Speed-Up	1	1.99	2.97	3.97	4.94	5.89	6.81	7.82	8.71	9.67	10.6	11.4	12.4	13.3	14.2	14.9
Effizienz	1	0.997	0.992	0.992	0.988	0.98	0.97	0.98	0.97	0.97	0.96	0.95	0.96	0.95	0.95	0.93



Man erkennt einen fast linearen Speedup bei einer Effizienz von knapp 1, mit anderen Worten ist das Problem recht gut parallelisiert worden. Die Berechnung an sich ist sogar ideal parallelisiert worden (jeder Wert wird nur einmal berechnet und nur einmal übertragen), doch je mehr Prozesse im System sind, desto mehr Zeit vergeht insbesondere beim Ringupdate, wodurch die Effizienz unter 1 gedrückt wird.

Gleichzeitig sollte nicht nur der Speedup des Programms gelobt werden. Schließlich ist auch eine hundertprozentige Effizienz ziemlich zwecklos, wenn die Grundgeschwindigkeit, gegenüber der die Effizienz gemes-

sen wird, zum weglaufen ist. Doch auch in diesem Sinne braucht sich das parallele Programm nicht zu verstekken, das mit genügend Prozessoren unter dem Hintern auch mit einer unfair schnellen Maschine wie der Sun mithalten kann; mit der vorgegebenen sequentiellen Implementation in `sunlife.c` allemal. Bei großen Bitmaps erweist sich einzig die Übertragung der X11-Bitmaps vom Transputer zum X-Server als Bremse, was sich allerdings nicht in Software beheben läßt.

Ein weiterer Pluspunkt, der noch erwähnt werden sollte, ist, daß der Speicherverbrauch der Gesamtmatrix auf die Einzelprozesse aufgeteilt ist. Zwar ist dies noch nicht implementiert, doch prinzipiell bräuchte der Master nur den Speicherplatz eines Slaves zum Empfangen und Weiterleiten einer Teilmatrix. Zusammen mit dem geringen Speicherverbrauch der Bitmaps lassen sich astronomisch große Spielfelder simulieren. Jeder Slave benötigt zur Pufferung zwei Felder (ließe sich bei Bedarf auf ein einziges reduzieren), d.h. bei 2 MB pro Prozessor lassen sich mindestens 620 kB für ein Feld allokieren. Auf 16 Prozessoren stünden 9.7 Megabytes zur Verfügung, was einem quadratischen Feld mit einer Kantenlänge von 9000 Pixeln entspricht. Der Forderung der Aufgabenstellung, möglichst große Felder zu ermöglichen, sollte damit Genüge getan sein.

Module

Hier eine kurze Auflistung der wichtigsten Module und Funktionen des Programms

Modul	***Funktion***	***Beschreibung***
`main.c`	`main ()`	Startpunkt des Masters. Macht nichts weiter als die Eingabeschleife aufzurufen.
`gui.c`	`MainLoop ()`	Haupteingabeschleife mit Eingabe des Benutzers. Aufgerufen vom `main()` des Masters.
	`Run ()`	Beginn einer Berechnung. Die Bildschirmausgabe wird initialisiert, und `master.c:Calc()` aufgerufen.
`master.c`	`Calc ()`	Berechnungsschleife. Die Prozessoren werden gezählt, die Bitmap aufgeteilt und verschickt. Anschließend sammelt eine Schleife so oft wie vom Benutzer gewünscht die Ergebnisse ein.
	`RingUpdate ()`	Wie der Name schon sagt. Ein Ringupdate (des zweiten Typs) wird durchgeführt. Das optimierte Ringupdate befindet sich in `Calc()`
`slave.c`	`main ()`	Der komplette Slave.
`lifecalc.c`	`BsNewGeneration ()`	Berechnung der nächsten Generation. Wird vom Slave aus aufgerufen.
`clcdl.c`		Erzeugung einer entsprechenden CDL-Datei.