Solitaire-Schach

— Praktikumsbericht —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Kira Duwe
E-Mail-Adresse:	0duwe@informatik.uni-hamburg.de
Matrikelnummer:	6225091
Studiengang:	Informatik

Vorgelegt von:	Enno Zickler
E-Mail-Adresse:	0zickler@informatik.uni-hamburg.de
Matrikelnummer:	6250134
Studiengang:	Informatik

Betreuer:	Julian Kunkel

Hamburg, den 16.10.2013

Abstract

Solitaire-Schach ist eine Solitairevariante, die ebenfalls allein aber nach Schachregeln gespielt wird. Ziel ist es, dass nur noch eine Figur auf dem Spielbrett übrig bleibt. In jedem Zug muss genau eine Figur geschlagen werden. Die Figuren sind normale Schachfiguren mit ihren üblichen Zugmöglichkeiten. Ausnahme ist nur der Bauer, da er hier in alle Richtungen ziehen kann. Die Startaufstellung auf einem 4x4 Schachbrett besteht aus einer beliebigen Kombination der folgenden Figuren: 1x Dame, 1x König, 2x Springer, 2x Läufer, 2x Turm und 2x Bauer.
Unser Programm errechnet, wie viele dieser Startpositionen lösbar sind. Dies ist für Spielbretter bis zur Größe von 21 Feldern (3x7) möglich. Da es für ein 4x4 großes Spielbrett bereits 6,7 Milliarden gültiger Startbelegungen gibt, kann unser Programm auch mit verschiedenen Parallelisierungsstufen auf einem Cluster aufgerufen werden. Parallelisiert wurde mit MPI und OpenMP.

Inhaltsverzeichnis

1 Einleitung

1.1 Begrifflichkeiten

Hier eine Definition der im folgenden gebrauchten Begriffe:

- Spielbrett: die aktuelle Belegung der einzelnen Felder
- Spielfeld: genau ein Einzelsegment des Spielbrettes
- Spielbrettbreite: x-Koordinate des Spielbrettes (links-rechts)
- Spielbretthöhe: y-Koordinate des Spielbrettes (oben-unten)
- Felderanzahl= Spielbrettbreite x Spielbretthöhe

1.2 Spielregeln

Solitaireschach ist eine Kombination aus der Spiellogik von Solitaire und den Zugregeln des Schachs. Es wird allein auf einem 4x4 großen Brett mit einer gegebenen Startaufstellung gespielt. Diese ergibt sich zufällig aus der Figurenmenge des Schachs, wobei nur zwei Bauern vorhanden sind. Es gibt also eine Dame, einen König, zwei Springer, zwei Läufer, zwei Türme und zwei Bauern. Zu Beginn kann eine beliebige Kombination dieser Figuren auf dem Brett stehen, wobei mindestens zwei und maximal zehn Figuren benutzt werden. Ungültig wären Spielbretter bei denen zum Beispiel zwei Damen aufgestellt sind.

Ziel ist es, dass am Ende des Spiels nur noch eine einzige Figur auf dem Spielbrett steht. Pro Zug muss genau eine Figur gezogen und damit eine andere geschlagen werden. Die Figuren ziehen nach folgenden Regeln:

- Bauer: Der Bauer kann auf eines seiner benachbarten Diagonalfelder ziehen. Im Gegensatz zum Schach ist hier jede Zugrichtung möglich.
- Turm: Der kann beliebig viele Felder (mindestens eines) auf der x- oder auf der y-Koordinate seines Feldes ziehen.
- Läufer: Der Läufer kann beliebig viele Felder (mindestens eines) auf den Diagonalen seines Feldes ziehen.
- Springer: Der Springer ist die einzige Figur, die über andere Figuren auf ihrem Weg hinwegspringen kann. Erst die Figur am Ende des Zuges wird geschlagen. Es gibt zwei mögliche Züge. Entweder der Springer wird zwei Felder auf der x-Koordinate

und eines auf der y-Koordinate gezogen oder ein Feld in x-Richtung und zwei in y-Richtung.

- König: Der König kann genau ein Feld weit ziehen, wobei alle benachbarten Felder möglich sind. Er stellt also eine Mischung aus Bauer und eingeschränktem Turm dar.
- Dame: Die Dame ist in der Lage beliebig viele Felder auf den Diagnoalen oder auf einer der beiden Achsen zu ziehen. Damit ist sie eine Kombination aus Läufer und Turm.

1.3 Aufgabenstellung

Das Spiel bietet nun eine enorme Vielfalt an gültigen Startpositionen und resultierenden Folgebrettern. Uns faszinierte daher die Frage, wie viele mögliche Startbelegungen es gäbe und welche davon lösbar wären. Dieses Problem wollten wir gern für eine variable Spielbrettgröße untersuchen und nicht nur für den Fall des 4x4-Brettes. Das Programm sollte also alle gültigen Startpositionen erzeugen - und zwar nur die gültigen - sowie eine Berechnung durchführen, welche dieser entstandenen Bretter lösbar sind. Sollten diese Aufgabenstellungen nicht umfangreich genug sein, so hätten wir uns noch mit der Ausgabe der möglichen Lösungen beschäftigt und einer Aufzeichnung der Lösungswege. Gegen die Ausgabe der verschiedenen Lösungswege entschieden wir uns, da der Bedarf an Speicherplatz jeglichen Rahmen gesprengt hätte.

2 Entwurf & Implementierung

2.1 Entwurf

2.2 Probleme

Während des Entwurfs und der Implementierung traten in einigen Bereichen Schwierigkeiten auf.

2.2.1 Spielbretterzeugung

Zunächst einmal bestand der Anspruch nicht nur einen korrekten sondern auch einen effizienten Algorithmus für die Erzeugung der Spielbretter auf einer variablen Spielbrettgröße zu finden. Es sollten also garantiert alle Spielbretter erzeugt werden, ohne dass doppelte entstehen.

2.2.2 Modellierung und Speicherung der Spielbretter

Eine obere Grenze für die Anzahl der möglichen Startbelegungen bietet eine Abschätzung mit der Faktoriellen:
16 * 15 * 14 * 13 * 12 * 11 * 10 * 9 * 8 * 7 = 29 059 430 400

Dort sind zwar noch die doppelten Felder enthalten, die dadurch entstehen, dass zwischen den Figuren vom gleichen Typ nicht unterschieden wird, dennoch bietet es eine gute Einschätzung der Größenordnung. Damit ergab sich die Frage, wie man die Spielbretter möglichst speichereffizient modellieren kann und wie eine sinnvolle Speicherverwaltung einer derart großen Menge aussehen könnte, die später auch noch für den parallelen Zugriff geeignet wäre.

2.2.3 Berechnung der Spielbretter

Um von einer gültigen Startbelegung zu der Aussage „Das Brett ist lösbar.“ zu kommen,braucht man mindestens einen im Maximalfall aber 9 Berechnungsschritte und zwar für jedes Spielbrett. Da die Zahl der möglichen Spielbretter ohnehin schon sehr groß ist, musste also eine Berechnung gefunden werden, die sehr effizient arbeitet.

2.2.4 Lastungleichheit

Aufgrund der Abhängigkeit der Berechnungszeit von der Anzahl der möglichen Züge ergibt sich eine deutliche Lastungleichheit. Es lässt sich nicht sagen, wie viele Figuren überprüft werden müssen, bis ein als lösbares gespeichertes Nachfolgebrett entsteht.

2.3 Implementierung

2.3.1 Darstellung der Spielbretter

Wir schwankten lange zwischen zwei verschiedenen Ansätzen. Eine Möglichkeit ist die Speicherung als zweidimensionales Array, das eine deutlich intuitivere Handhabung mit sich bringt. Allerdings ist es deutlich größer, was den benötigten Speicher angeht. Das kleinste zur Verfügung stehende wäre ein 16 elementiges Char-Array. Es braucht also für den Fall 4x4: 16*8bit = 128 bit.

Um eine möglichst kompakte und damit speicher- und zugriffseffiziente Darstellung der Spielbretter zu erhalten, entschieden wir uns für eine Oktaldarstellung. Diese benötigt 64 bit und bietet die Möglichkeit, Spielbretter bis zu einer Größe von 3 x 7 zu codieren. Das 4x4 Brett braucht also nur 64 bit und damit die Hälfte im Vergleich zum Array.

Codierung:
Die letzte Stelle der Zahl ist das Feld des Brettes mit der Nummer 0. Jede Stelle der Oktalzahl bzw. drei Stellen der Binärzahl ($2^3 = 8$) repräsentiert also genau ein Feld des Spielbrettes. Für die Spielfiguren führten wir folgende Codierung ein:

- 0 = leeres Feld
- 1 = Bauer
- 2 = Turm
- 3 = Läufer
- 4 = Springer
- 5 = König
- 6 = Dame

Die Felder des Spielbrettes nummerierten wir von 0 - Felderanzahl-1 durch.

Hier ein Beispiel wie für die Darstellung eines Spielbrettes als Oktalzahl. Das Spielbrett: 3032 4124 0650 0010 entspricht dem in der Tabelle dargestellten.

Diese Art der Repräsentation orientiert sich an dem gängigen Standard für die Speicherung von Schachbrettern. Dort wird jedoch meistens für jeden Figurentypen eine Binärzahl vorgehalten, weil so ein ganzes Brett in 64 Bit passt. Da wir aber ein deutlich kleineres Spielbrett betrachten, entschieden wir uns dafür nur eine Zahl zu benutzen.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Tabelle 2.1: Nummerierung der Spielfelder

0	1	0	0
0	5	6	0
4	2	1	4
2	3	0	3

Tabelle 2.2: Spielbrett 3032 4124 0650 0010

2.3.2 Erzeugung der Spielbretter

Die Erzeugung erfolgt über ineinander verschachtelte for-Schleifen. Der naive Ansatz zählt für jede Figur jeweils von 0 bis Spielbrettgröße. Da aber zwischen den doppelten Figuren (Springer, Läufer, Turm, Bauer) nicht unterschieden wird, ist es möglich an dieser Stelle zu optimieren. So beginnt die jeweils zweite Figur die for-Schleifen erst ab der Position der jeweils ersten Figur zu durchlaufen, da ansonsten ein bereits erzeugtes Spielbrett erneut durchlaufen würde. Eine weitere Optimierung ist möglich, indem alle weiter innen liegenden Schleifendurchläufe abgeschnitten werden, sobald das betrachtete Feld nicht frei ist. Denn es wird unabhängig vom Typ der Figur niemals möglich sein, diese auf ein bereits belegtes Feld zu setzen.

Listing 2.1: Erzeugung der Spielbretter

```
1  for(2 bis maximaleFigurenanzahl){
2  //Schleife Dame
3  for(posDame = 0; posDame <= anzahlFelder; posDame++){
4      if(positionDame<anzahlFelder){
5           setze Dame;
6           anzahlFiguren ++;
7      }
8
9      //Schleife Koenig
10     for(posKoenig = 0; posKoenig <= anzFelder; posKoenig++){
11          // uebernehmen der bisher gesetzen Figuren
12          spielbrettKoenig = spielbrettDame;
13          // uebernehmen der bisherigen Figurenanzahl
14          anzFigurenK = anzFigurenD;
15
16          if(!(anzFigurenK < maxFiguren)){
```

```
17                setze keinen Koenig;
18           }
19
20           if(feldFrei(posKoenig, spielbrettKoenig)){
21                if(posKoenig<anzFelder){
22                     setze Koenig;
23                     anzahlFiguren ++;
24                }
25
26                //Schleife 1. Springer
27                for(posSpringer1 <= anzFelder){
28                      ...
29                      if(!(anzFigurenS < maxFiguren))
30                      ...
31                      if(feldFrei){
32                          if(posSpringer < anzFelder)
33                          ...
34                          posSpringer2=posSpringer1;
35                          //Schleife 2. Springer
36                          for( posSpringer2<=anzFelder){
37                          ...
38                           //Schleifen 1.+ 2. Laeufer
39                           ...
40                           //Schleifen 1. + 2. Turm
41                           ...
42                           //Schleife 1. Bauer
43                           ...
44                           //Schleife 2. Bauer
45                           for(Bauer2) {
46                              if(! <)
47                              if(feldFrei){
48                                   if(<){
49                                      setze Bauer2;
50                                      berechneSpielbrett;
51                                   }
52                              }
53                          }
54                      }
55                  }
56              }
57          }
58      }
59  }
```

Die äußere for-Schleife sorgt dafür, dass erst alle Spielbretter mit zwei Figuren erzeugt werden und danach aufsteigend alle mit mehr Figuren. Dies ist notwendig, da die Berechnung auf der aufsteigenenden Figurenanzahl aufbaut und in der innersten Schleife direkt aufgerufen wird. Das ermöglicht die Verwendung eines Sets zur Speicherung. (siehe Unterkapitel Speicherung der Spielbretter)

2.3.3 Berechnung der Spielbretter

Um die bereits berechneten Lösungen weiter zu verwenden, findet die Berechnung in Abhängigkeit der auf dem Brett befindlichen Figurenanzahl statt. Zunächst werden alle Bretter mit einer Figur als lösbar vermerkt. Es findet nun eine ebenenweise Berechnung statt, bei der nach der Figurenanzahl aufgestiegen wird. So kann für jedes Spielbrett einer höheren Ebene überprüft werden, ob daraus ein lösbares Brett entstehen kann. Der Ansatz der dynamischen Programmierung ist eine erhebliche Optimierung, da ohne diese Wiederverwendung im schlechtesten Fall noch acht weitere Berechnungen stattfinden müssten.

Für die Berechnung werden die Spielbretter nicht nur in der Oktaldarstellung sondern auch als Array vorgehalten. Das hat den Vorteil, dass die Abfragen zur Überprüfung ob, eine Spielfigur sich nach einem Zug auch noch auf dem Spielbrett befindet, sehr viel übersichtlicher und verständlicher werden. Denn eine für verschiedene Spielbrettgrößen passende Abfrage ist auf Oktalzahlen zu aufwändig.

Es findet also zunächst eine Umwandlung in die Arraydarstellung statt. (siehe Listing 2.2)

Listing 2.2: Umwandlung in Arraydarstellung

```
1  // Berechnung der Arraydarstellung aus Oktaldarstellung
       ↪ einfacher fuer die
2  // Ueberpruefung der Spielbrettgrenzen*/
3  for(x=0; x < SpielbrettBreite; x++)
4  {
5      for(y=0; y < SpielbrettHoehe; y++)
6      {
7          param.spielbrett_array[x][y] =
8          (spielbrett >> ((x+(y*SpielbrettBreite)) * 3)) % 8;
9          }
10     }
```

Das daraus entstandene zweidimensionale Array wird nun also Feld für Feld überprüft, ob dort eine Figur steht. Ist dies der Fall wird berechnet, ob es für diese Figur mögliche Züge gibt, die zu einem lösbaren Brett führen. (siehe Listing 2.3)

Listing 2.3: Berechnung eines Spielbrettes

```
1  for(x=0; (x < SpielbrettBreite) && (loesbar == 0); x++)
2  {
3      for(y=0; (y < SpielbrettHoehe) && (loesbar == 0); y++)
4      {
5          switch(param.spielbrett_array[x][y]){
6              case DarstellungBauer:
7                  loesbar= berechneBauer(&param, x, y);
8                  break;
9              case DarstellungTurm:
10                 loesbar = berechneTurm(&param, x, y);
11                 break;
12             case DarstellungSpringer(&param, x, y);
13                 break;
14             case DarstellungKoenig:
15                 loesbar = berechneKoenig(&param, x, y);
16                 break;
17             case DarstellungDame:
18                 loesbar = berechneDame(&param, x, y);
19                 break;
20         }
21     }
22 }
```

Das Schlagen einer Figur geschieht auf der Darstellung als Oktalzahl. Um die schlagende und die geschlagene Figur vom Brett zu löschen, wird die Oktalzahl mit einer Bitmaske aus Einsen verundet, die an den beiden entsprechenden Positionen eine 0 hat. Anschließend wird die Figur an die neue Position geschrieben.

(siehe Listing 2.4)

Listing 2.4: Schlagen einer Figur

```
1  void schlageFigur()
2  {
3  einser_Bitmaske = 0xffffffffffffffLL;
4
5  // Spielfiguren von Spielbrett loeschen
6  // Von der Bitmaske wird "(7 << pos*3)" abgezogen, um an
       ↪ dieser Stelle 0 zu erzeugen
7      neues_spielbrett = spielbrett &
8      (einser_Bitmaske - (7 << pos*3) - (7 << neue_pos*3));
9
10     // nach Schlagen Spielfigur neu  setzen
11     neues_spielbrett += (DarstellungFigur << neue_pos*3);
12 }
```

2.3.4 Speicherung der Spielbretter

Die ursprüngliche Speicherung der Spielbretter erfolgte nach der Figurenanzahl in verschiedenen Hashtabellen. In diesen wurde hinterlegt, ob ein Spielbrett lösbar ist (1) oder nicht (0). Verwendet wurde hier ein directhash, da die Spielbrettrepräsentation bereits eindeutig ist. Aufgrund der enorm großen Anzahl an gültigen Spielbrettern ist das Speichern all dieser 64bit-Integer aber nicht ohne weiteres möglich.

Also optimierten wir die Speicherung. Der nächste Schritt war die Verwendung der Hashtabelle als Set. Es wurden also nur noch die lösbaren Bretter gespeichert. Damit entfiel die Speicherung des Wertes, der aufgrund der Signaturen der verwendeten Bibliothek (glib) ebenfalls 64 bit groß gewesen wäre, obwohl wir nur ein Bit benötigt hätten. Dies war jedoch erst dadurch möglich, dass wir die Erzeugung umbauten, sodass sie die Spielbretter mit zwei Figuren zuerst erzeugte und direkt in der innersten Schleife die Berechnung stattfand.

Im Laufe der Analyse stellte sich heraus, dass die Menge an lösbaren Bretter deutlich größer ist als die Menge der nicht lösbaren. Es gibt sogar Figurenanzahlen, bei denen alle Spielbretter lösbar sind(siehe Kapitel 3). Daher wird die Hashtabelle inzwischen als Set für die Speicherung der nicht lösbaren Spielbretter verwendet.

Anzahl lösbare Bretter bei 4x4: 6.683.363.975
Anzahl nicht lösbare Bretter bei 4x4: 1.047.721

Anfangs war folgender Speicherbedarf notwendig für ein 4x4-Spielbrett:
6.684.411.696 x 64bit x 2 = 855.604.697.088 bit = 106 GByte
Inzwischen sind es aber nur noch:
1.047.721 x 64 bit = 67.054.144 bit = 8,3 MByte

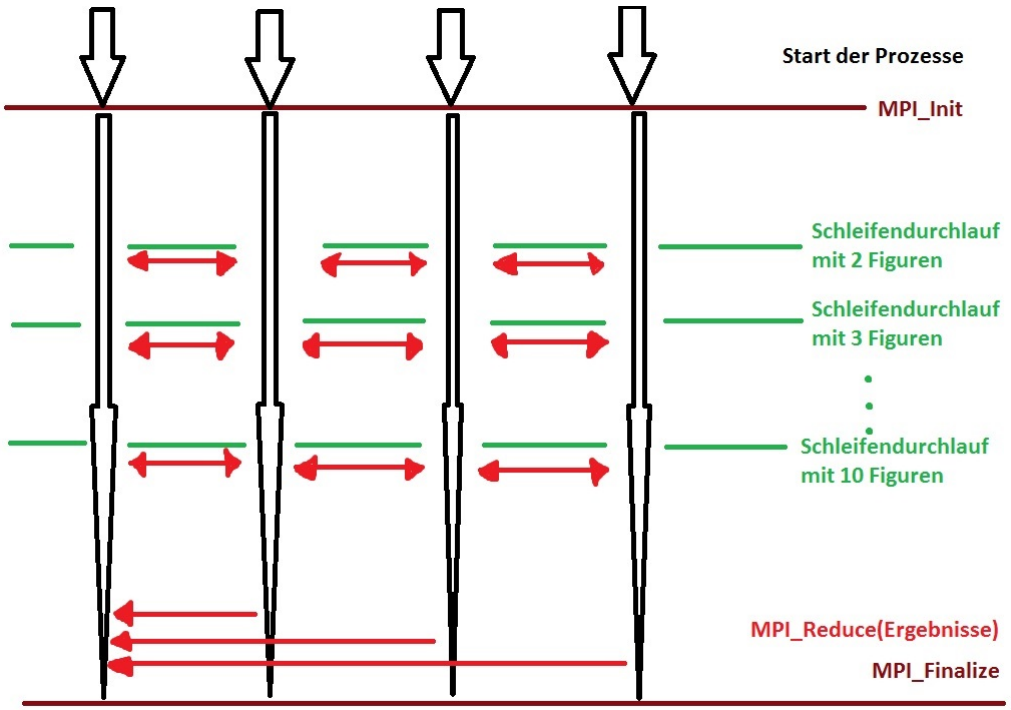
Eine weitere Optimierung hat dadurch stattgefunden, dass nur noch die aktuellen Bretter und die aus der vorherigen Ebene gespeichert werden und der Rest verworfen wird. Denn für die Berechnung ist jeweils nur das aktuelle Brett und die Information, welche Bretter bisher lösbar waren, notwendig.

2.3.5 Parallelisierung

Für die Parallelisierung haben wir sowohl MPI als auch OpenMP verwendet. Beides ließ sich sehr gut auf unsere verschachtelten for-Schleifen anwenden. Wir haben die äußerste Schleife (die der Dame) für die MPI Aufteilung verwendet. Damit ergibt sich hier als Grad der maximalen Parallelität die Anzahl der Felder + 1. Dies reichte für unseren Cluster jedoch vollkommen, da wir für ein 4x4 und sogar für 3x3 Bretter alle 10 Knoten benutzen konnten. Um aber auch alle Kerne auf den Knoten zu benutzen, haben wir die nächsten beiden Schleifen mit OpenMP parallelisiert. Umgesetzt wurde dies mit der nested Funktion von OpenMP. Die Benutzung des collapse-Befehls war aufgrund der Abbruchbedingung, ob das Feld schon belegt ist, nicht möglich.

2.3.6 Kommunikation

Der eigentlich Knackpunkt war nach der Aufteilung auf die Knoten die Kommunikation zwischen ihnen. Erst die Speicherplatzoptimierung und vorallem die Möglichkeit, die Spielbretter in der Reihenfolge der Anzahl ihrer Spielfiguren zu erzeugen, machte uns eine eine effiziente Kommunikation möglich. Wenn wir alle Spielbretter hätten vorhalten müssen, dann wäre es uns nicht möglich gewesen, jedem Knoten auf seinem Speicherplatz alle Spielbretter zur Verfügung zu stellen. Somit hätte es bei jedem Nachgucken, ob ein Spielbrett lösbar ist, zu einer Kommunikation kommen können und in vielen Fällen sogar müssen, da dieser Teil der Daten auf einem anderen Knoten liegt. Durch die Minimierung des Speicherbedarf war es uns jedoch möglich, nach der Berechnung für eine Figurenanzahl die Daten aus dem letzten Berechnungsdurchlauf von jedem Knoten an jeden anderen zu senden, sodass für den nächsten Durchlauf jeder Prozess auf alle vorherigen Ergebnisse zurückgreifen kann.
Hierzu war es nötig, dass alle Prozesse nach einem Durchlauf zunächst über eine MPI-Barrier synchronisiert werden. Anschließend musste jeder Prozess den anderen mitteilen, wie viele Spielbretter er im letzen Durchlauf als nicht lösbar gespeichert hat und den anderen senden möchte. Daraufhin wird in jedem Prozess ein Buffer für die erwarten Spielbretter alloziert. Außerdem muss der Inhalt des eigenen Hashsets in ein Buffer zum Senden kopiert werden. Zusätzlich werden noch die Zähler für die Anzahl der errechneten Spielbretter vom Master-Prozess eingesammelt, um sie am Ende ausgeben zu können.



3 Ergebnisse

```
Berechnungszeit:    191.835981 s
Bretter / Sekunde: 34844430.148899

Figuren:  Loesbare:    Gesamt:  % Lösbar:  Zeit Max    Min     Avg
================================================================================
  0              0           0      -nan      0.00     0.00     0.00
  1             96          96    100.00      0.00     0.00     0.00
  2           4482        7920     56.59      0.02     0.01     0.01
  3          71521      100800     70.95      0.05     0.02     0.03
  4        1425531     1594320     89.41      1.58     0.18     0.56
  5       16373752    16773120     97.62      9.25     4.41     5.74
  6      117846720   118198080     99.70     28.10    13.09    16.25
  7      547652219   547747200     99.98     59.50    39.19    43.58
  8     1589184012  1589187600    100.00     61.96    47.42    59.09
  9     2594592000  2594592000    100.00     46.06    25.67    40.58
 10     1816214400  1816214400    100.00     22.31     8.35    19.83
================================================================================
Summe:  6683363975  6684414768     99.98    191.84
```

Anzahl der gültigen Spielbretter für ein 4x4-Brett = 6.684.411.696 :
Anzahl nicht lösbare Bretter bei 4x4: 1.047.721

Der Anteil an nicht lösbaren Spielbrettern wird bei größerer Spielbrettgröße immer kleiner. Das hängt damit zusammen, ob eine Dame auf dem Brett platziert ist oder nicht. Je mehr Figuren auf dem Brett stehen, umso größer ist die Wahrscheinlichkeit, dass sich auch die Dame dort befindet. Und die Wahrscheinlichkeit, dass ein Brett lösbar ist, wächst mit der Zahl der aufgestellten Figuren, wenn auch eine Dame gesetzt ist. Das liegt an der großen Vielfalt an möglichen Zügen, die eine Dame hat. So sind die Spielbretter mit zehn Figuren in der Größe 4x4 immer lösbar.

Wir können an dieser Stelle keinen formalen Beweis für die Korrektheit der Behauptung anführen, da jedoch die Menge der nicht lösbaren Bretter gleich blieb, unabhänig davon ob diejenigen mit zehn Figuren mitbetrachtet wurden oder nicht, brauchen diese also nicht mehr berechnet werden.

3.1 Analyse

Im Profiling des Programms kann man sehr schön sehen, auf welchen Funktionen unsere Berechnung sich am stärksten stützt. Diese sind vorallem die Abfrage bei der Erzeugung, ob ein Feld frei ist und in der Berechnung die Funktion zum Schlagen einer Figur. Bei diesen beiden Funktionen haben wir deshalb eine möglichst effiziente Implementierung

gewählt. Es zeigt sich aber auch, dass das Umrechnen in ein Array viel Zeit kostet und man hier für weitere Optimierungen doch noch einmal versuchen könnte, die Berechnung vollständig auf den Oktalzahlen zu implementieren.

Gprof des sequentiellen Programms:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 32.51     11.75    11.75 181063344     0.00     0.00  spielbrettBerechne
 28.54     22.06    10.31        1    10.31    36.02  spielbretter_berechne
 11.15     26.08     4.03 1606358325     0.00     0.00  feldFrei
  8.52     29.16     3.08 181063344     0.00     0.00  spielbretterArrayCreate
  5.76     31.24     2.08 181063344     0.00     0.00  spielbretterArrayDestruct
  3.84     32.63     1.39 183252731     0.00     0.00  schlageFigur
  1.70     33.24     0.62 39687589      0.00     0.00  berechneSpringer
  1.56     33.81     0.57 39762256      0.00     0.00  berechneLaeufer
  1.54     34.36     0.56 39927693      0.00     0.00  berechneTurm
  1.52     34.91     0.55 183252731     0.00     0.00  neuesSpielbrettLoesbar
  1.15     35.33     0.42 39052394      0.00     0.00  berechneBauer
  1.02     35.70     0.37 20782134      0.00     0.00  berechneKoenig
  0.86     36.01     0.31 20732220      0.00     0.00  berechneDame
  0.26     36.10     0.10                             main
  0.07     36.13     0.03                             frame_dummy
  0.04     36.14     0.02        1     0.02     0.02  erzeugeHashtables
  0.00     36.14     0.00        1     0.00     0.00  AskParams
  0.00     36.14     0.00        1     0.00     0.00  gibStatistikAus
  0.00     36.14     0.00        1     0.00     0.00  spielbretterErzeugung1Figur
```

Profiling von vampiretrace des parallelen Programms:

```
                                excl. time  incl. time
*excl. time  incl. time      calls     / call     / call  name
    0.601s      0.601s        19.92    30.201ms    30.201ms  !$omp ibarrier @spielbretter.c:545
   40.338ms     0.619s        20.92     1.929ms    29.617ms  !$omp for @spielbretter.c:329
   26.816ms    26.816ms        0.23     0.117s      0.117s   MPI_Barrier
   18.551ms     3.010ms     7560.33     2.454us     0.398us  spielbretterArrayCreate
   11.405ms     1.138ms     8549.56     1.334us     0.133us  schlageFigur
   10.480ms     2.200ms     7559.96     1.386us     0.291us  spielbretterArrayDestruct
    9.443ms     4.904ms    37300.17     0.253us     0.131us  feldFrei
    9.059ms     2.196ms     2110.31     4.292us     1.040us  berechneDame
    8.904ms    19.387ms     7560.35     1.177us     2.564us  spielbrettBerechne
    7.608ms     1.708ms     1986.56     3.830us     0.860us  berechneSpringer
    7.172ms     7.172ms        0.92     7.824ms     7.824ms  MPI_Bcast
    3.602ms     3.602ms     8549.42     0.421us     0.421us  neuesSpielbrettLoesbar
    2.355ms    58.659ms        1.23     1.916ms    47.722ms  parallel region
    1.986ms     1.199ms     1347.17     1.474us     0.890us  berechneTurm
    1.942ms     1.008ms     1466.42     1.324us     0.687us  berechneLaeufer
    1.782ms     0.802ms     1410.73     1.263us     0.568us  berechneBauer
    0.757ms     0.837ms        0.04    18.172ms    20.093ms  MPI_Init
    0.641ms     0.000ns        0.04    15.373ms     0.000ns  spielbretter_berechne
    0.565ms     1.272ms     1378.92     0.409us     0.922us  berechneKoenig
    0.251ms     0.000ns        0.04     6.030ms     0.000ns  user
    0.166ms     0.266ms      259.21     0.640us     1.027us  !$omp critical @spielbretter.c:209
    0.100ms     0.100ms      259.21     0.387us     0.387us  !$omp critical sblock @spielbretter.c:210
   80.032us    80.032us        0.04     1.921ms     1.921ms  sync time
   19.662us     0.620s        20.92     0.940us    29.618ms  !$omp parallel @spielbretter.c:329
    3.786us     0.000ns        0.04    90.873us     0.000ns  main
    2.256us     2.256us        0.04    54.148us    54.148us  erzeugeHashtables
    0.661us     0.661us        0.04    15.856us    15.856us  AskParams
    0.199us     0.199us        0.08     2.394us     2.394us  MPI_Initialized
   75.743ns    75.743ns        0.04     1.818us     1.818us  spielbretterErzeugung1Figur
   38.246ns    38.246ns        0.04     0.918us     0.918us  MPI_Comm_size
   12.374ns    12.374ns        0.04     0.297us     0.297us  MPI_Comm_rank
    0.000ns     0.000ns           1     0.000ns     0.000ns  tracing off
```

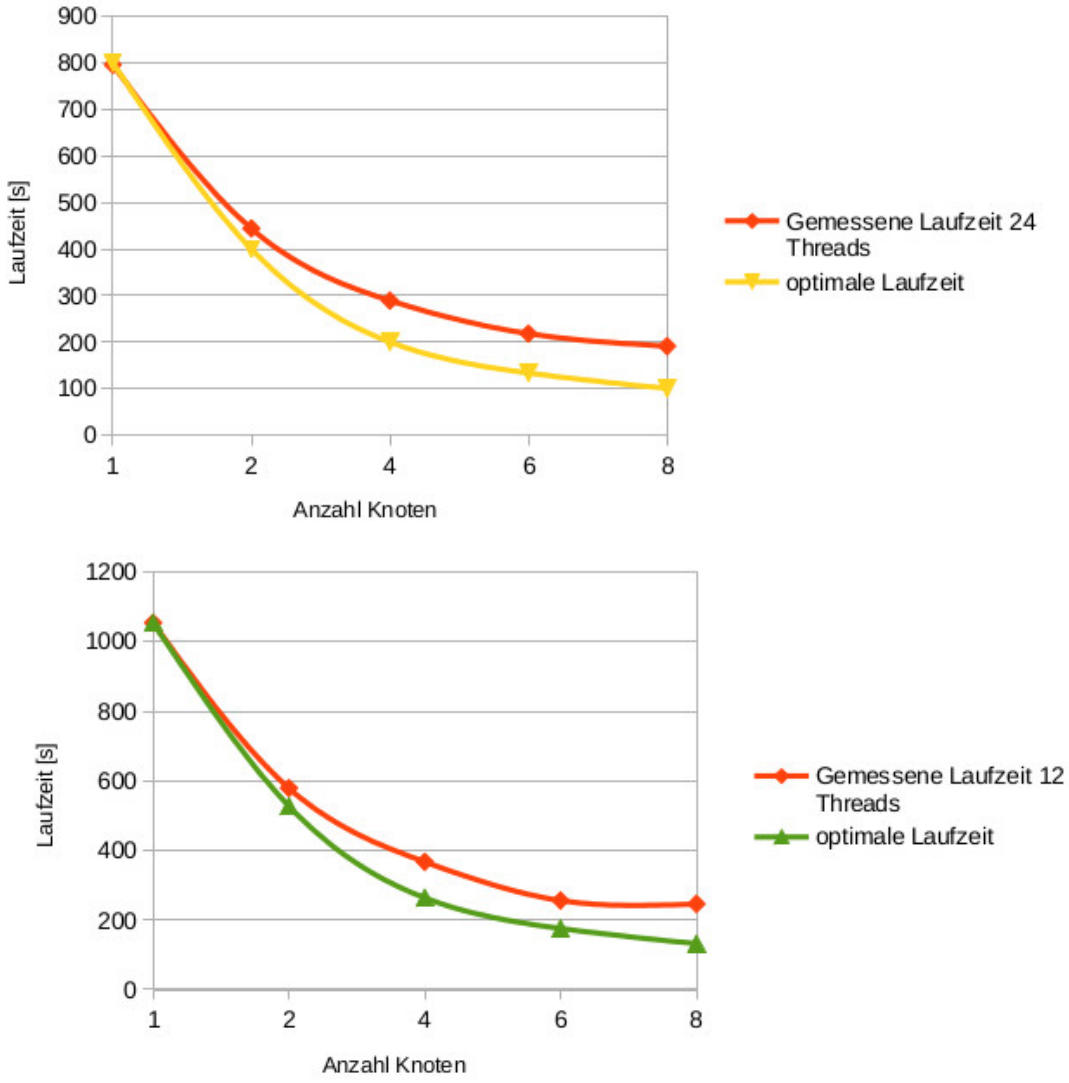
Im Profiling der parallelisierten Variante ist auch zu sehen, dass eine gewisse Lastungleichheit eine Menge Zeit kostet. Hierbei scheint die Aufteilung bei OMP jedoch noch schlechter zu sein als bei MPI. Schön ist allerdings, dass der parallele Zugriff auf die glib Hashtabelle, die einen kritischen Abschnitt für OpenMP darstellt, nicht so sehr ins Gewicht fällt. Dies war insbesondere bei den ersten sequentiellen Programmen ein ganz großes Problem.

Sehr erfreulich finden wir auch unsere Ergebnisse hinsichtlich Laufzeit, Speicherbedarf und Skalierbarkeit. Wie zuvor schon erwähnt, haben wir die Laufzeit von unserem naivem Ansatz mit einer geschätzten Laufzeit von 17 Tagen auf eine Laufzeit von 93 min und in der parallelen Variante auf 191 sec reduziert. Auch bei dem Speicherbedarf sind wir von erwarteten 106 GByte auf eine Speicherbedarf von unter 10 MByte gekommen. Und auch die Skalierbarkeit kann sich durchaus sehen lassen. Ihre Grenzen sind zwar abhängig von der Spielbrettgröße, es ist jedoch auch fraglich, ob sich ein weiterer Performancegewinn noch erzielen ließe, solange kleine und auch große Spielbretter betrachtet werden. Denn auf beispielsweise einem 2x3 Spielbrett würde die Kommunikation einen erheblichen größeren Anteil ausmachen als die eigentliche Berechnung. Damit bringt eine schlichte Erhöhung der Knotenanzahl und Threadanzahl vermutlich kaum noch Nutzen.

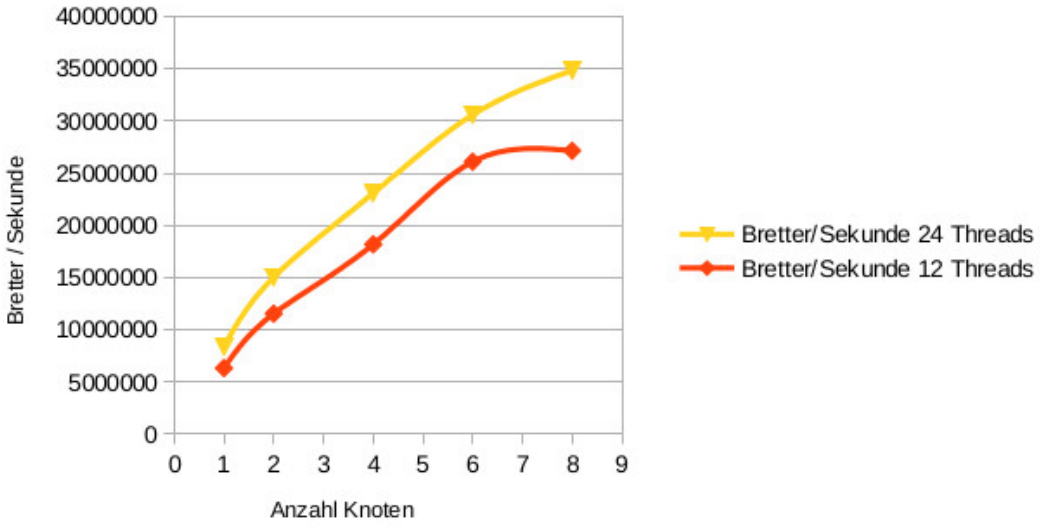
3.2 Speedup

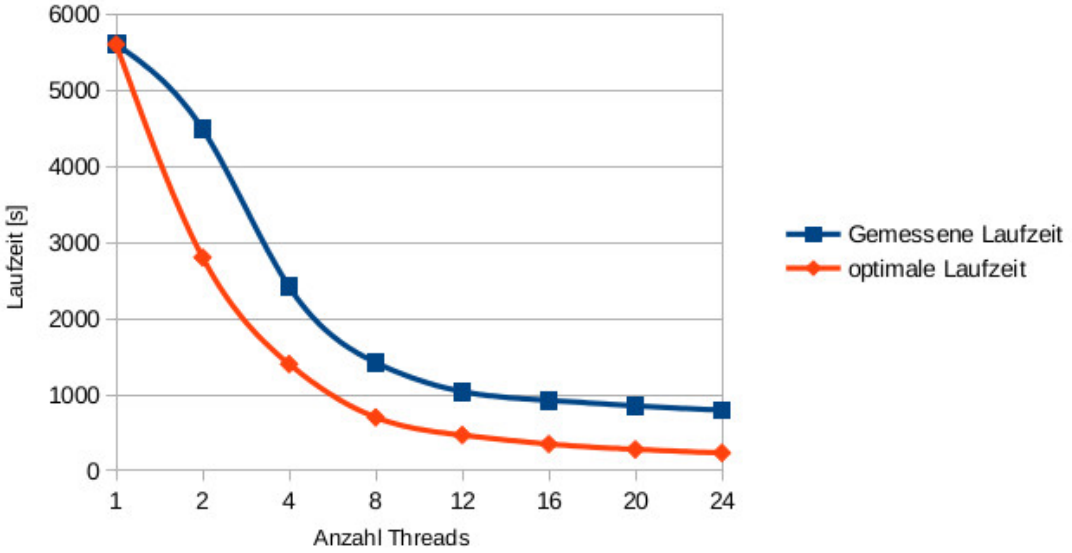
In den ersten beiden Diagrammen läst sich gut erkennen, dass der Speedup durch MPI sehr gut ist, aber mit zunehmender Prozessanzahl der Kommunikationsaufwand steigt und sich der Speedup weiter vom Optiumum entfernt. Hinzu kommt sicher auch die nicht ganz optimale Lastverteilung. Der Speedup durch OpenMP ist hingegen nicht ganz so gut wie bie MPI. Dies wird an der Lastungleichheit und an dem gemeinseam Zugriff auf die Hashtabellen liegen. Trotzdem haben wir auch hiermit ein deutlichen Performancegewinn erzielt.

Speedup mit MPI:



Speedup mit OpenMP:





4 Fazit

Das Praktikum hat wie kaum eine andere Veranstaltung gezeigt, worauf es bei der praktischen Lösung eines Problems ankommt.

Wir haben Fehler gemacht, wie zum Beispiel „einfach drauf los“ zu programmieren und erst nach und nach ein Konzept für das Programm zu entwickeln. Dadurch dass wir zu Beginn keine konkrete Vorstellung hatten, wo wir eigentlich hinwollten, war es schwierig einen passenden Zeitplan zu erstellen. Inzwischen sind wir also um viele sehr wertvolle Erfahrungen reicher und wissen, wie man bei Null anfangen sollte. Zum ersten Mal im Studium sollten wir ein selbst gewähltes Problem modellieren und auch lösen. Unsere Problemwahl haben wir nie bereut, denn Solitaire-Schach bietet viele Möglichkeiten, um sich als Informatiker damit auseinander zu setzen und hat uns sehr gefesselt. Die Entwicklung eines Algorithmus, der alle gültigen Startbretter erstellt und zwar nur diese, war ein sehr faszinierender und auch anspruchsvoller Prozess. Sie erforderte zu Beginn einiges an Abstraktionsvermögen, um mit der gewählten Oktaldarstellung erfolgreich arbeiten zu können. Auch die Notwendigkeit für Optimierungen forderte uns heraus. Zunächst zeichnete sich für das sequentielle Programm eine Laufzeit von 17 Tagen für ein 4x4-Spielbrett ab. Diese haben wir inzwischen auf ungefähr eineinhalb Stunden reduzieren können. Wir haben also die Aufgabenstellung erfüllt, die wir uns selbst gestellt haben und sind zufrieden mit unseren Ergebnissen. Unsere Kenntnisse in der C-Programmierung sind erheblich größer und detaillierter, als sie es zu Beginn waren. Gleiches gilt für die Parallelisierung mit MPI und OpenMP. Die Arbeit auf dem Cluster war sehr lehreich und auch über Jobskripte wissen wir nun einiges mehr. Schlussendlich war es sehr hilfreich mit git zu arbeiten.

Kaum eine andere Veranstaltung war so anspruchsvoll und fordernd und zeitgleich so unglaublich spannend wie das Praktikum. Die kompetente und freundliche Betreuung hat es zu einem tollen Projekt gemacht.

Einzig der späte Beginn mit der praktischen Umsetzung des eigenen Aufgabe war schade, denn wir hätten gern noch den anderen Lösungsansatz für unser Problem gebaut. Durch die knappe Zeit am Ende haben wir den mit Julian Kunkel entwickelten „Bottum-up-Ansatz“ aber leider nicht weiter verfolgen können, sonst wäre für jegliche Analyse keine Zeit mehr geblieben.