# Implementierung und Parallelisierung von Peg-33

Florian Ehmke

3. April 2011

# Inhaltsverzeichnis

**1 Einleitung** **2**
1.1 Spielregeln . . . . . . . . . . 2
1.2 Aufgabenstellung . . . . . . . . . . 2

**2 Design und Implementierung** **3**
2.1 Probleme . . . . . . . . . . 3
2.1.1 Spielzüge und Spielfeld effizient implementieren . . . . 3
2.1.2 Tiefensuche nicht praktikabel . . . . . . . . . . 3
2.1.3 Zuordnung Spielfeld zu Prozess . . . . . . . . . . 3
2.1.4 Verwaltung der Spielfelder und Lösungen . . . . . . . 3
2.1.5 Sehr große Zahlen . . . . . . . . . . 3
2.1.6 Lastungleichheit . . . . . . . . . . 4
2.2 Implementierung . . . . . . . . . . 4
2.2.1 Spielfeld und Spielzug . . . . . . . . . . 4
2.2.2 Algorithmus . . . . . . . . . . 5
2.2.3 Zuordnung Spielfeld zu Prozess . . . . . . . . . . 6
2.2.4 Verwaltung der Spielfelder und Lösungen . . . . . . . 7
2.2.5 Kommunikation . . . . . . . . . . 7
2.3 Ergebnisse . . . . . . . . . . 9
2.3.1 Laufzeiten . . . . . . . . . . 9
2.3.2 Lösungen / Sekunde . . . . . . . . . . 10
2.3.3 Speedup . . . . . . . . . . 10
2.3.4 Hashtablesize . . . . . . . . . . 10
2.3.5 Tracing . . . . . . . . . . 14

**3 Zusammenfassung** **15**

**Literatur** **16**

# 1 Einleitung

## 1.1 Spielregeln

Das klassische Peg-33 (auch Solitär, Solohalma oder Hi-Q genannt) besteht aus einem Spielfeld mit 33 Löchern. Zu Beginn sind bis auf eines (die Startposition) alle Löcher mit einem Steinchen (Peg[1] genannt) gefüllt. Das Ziel ist, durch das Setzen der Pegs ein Spielfeld zu erreichen, in dem nur noch ein Peg übrig ist – und zwar genau in der Startposition. Beim Setzen wird mit einem Peg immer ein Feld übersprungen. Um ein Peg setzen zu dürfen muss die Zielposition leer sein und in dem übersprungenen Feld muss sich ein Peg befinden. Das übersprungene Steinchen wird anschließend entfernt. Man benötigt also 31 Züge um zu einem Spielfeld zu gelangen, in dem nur noch ein Peg übrig ist.

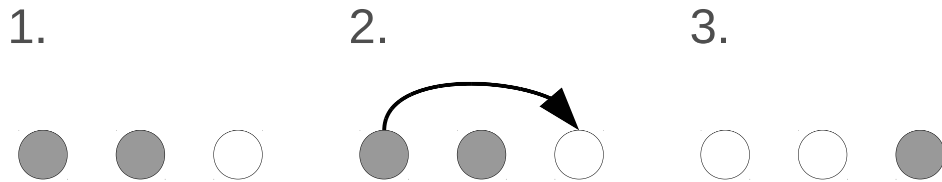


Abbildung 1: Setzen eines Pegs

## 1.2 Aufgabenstellung

Das Spiel Peg-33 lässt sich auf 40,861,647,040,079,968 verschiedenen Wegen lösen. Es sollte ein Programm entworfen und implementiert werden, welches alle möglichen Lösungen für ein Solitär-Spiel zählt, also im Falle von Peg-33 mit Startposition (3,3 – in der Mitte des Spielfeldes) auf die o.g. Zahl kommt. Weiterhin sollte es möglich sein verschiedene Spielfelder anzugeben und von verschiedenen Positionen zu starten.

[1] engl. für Stift/Dübel

# 2 Design und Implementierung

Eine Lösung für Solitär zu finden ist ein klassisches Suchbaum Problem.

## 2.1 Probleme

Einige Punkte während der Planung und Implementierung des Programmes beanspruchten deutlich mehr Zeit als andere.

### 2.1.1 Spielzüge und Spielfeld effizient implementieren

In Anbetracht der enormen Größe des Suchbaumes ist es nötig, das Spielfeld, und auch einen Spielzug auf diesem, effizient zu modellieren. Die Operationen auf dem Spielfeld – prüfen ob ein Zug möglich ist, einen Zug durchführen oder rückgängig machen – bilden den Kern des Algorithmus und werden am häufigsten ausgeführt.

### 2.1.2 Tiefensuche nicht praktikabel

Während der Implementierung einer ersten Version stellte sich heraus, dass einfache Tiefensuche („Bruteforce") viel zu ineffizient und außerdem schlecht parallelisierbar ist. Das Verfahren findet zwar recht schnell erste Lösungen, aber um *alle* Lösungen zu finden ist es viel zu langsam.

### 2.1.3 Zuordnung Spielfeld zu Prozess

Es ist nötig ein Spielfeld einem Prozess eindeutig zuzuordnen. Problematisch war, eine Zuordnung zu finden, bei der *alle* Spielfelder einem Prozess zugeordnet sind.

### 2.1.4 Verwaltung der Spielfelder und Lösungen

Mit zunehmendem voranschreiten im Suchbaum steigt die Anzahl der gespeicherten Spielfelder und dazugehörigen Lösungen (siehe Abbildung: 5). Elementare Operationen wie das wiederfinden eines Spielfeldes oder das erhöhen eines einem Spielfeld zugeordneten Zählers mussten schnell durchführbar sein.

### 2.1.5 Sehr große Zahlen

Die Anzahl der Lösungen, Spielfelder sowie das Spielfeld selbst benötigen wenigstens 64 Bit große Variablen. Bei einigen Konstellationen reicht selbst das nicht aus. Beispielsweise Peg-33 mit Startposition (2,4):

- `138,409,681,956,904,365,268` Lösungen [2]
- `18,446,744,073,709,551,615 = maxValue(uint64)`

### 2.1.6 Lastungleichheit

Die Zuordnung von Spielfeld ↔ Prozess erreicht nur eine ungleiche Verteilung. Daraus, und aus der Beschaffenheit der Spielfelder, entsteht eine Lastungleichheit. Selbst wenn alle Prozesse die gleiche Anzahl Spielfelder hätten, wäre die Last nicht ausgeglichen, da einige Spielfelder deutlich mehr Züge erlauben als andere.

## 2.2 Implementierung

Das Programm ist komplett in C (Standard c99) geschrieben. Es wurden die Bibliotheken Glib-2.0 (Hashtables) und MPI (Parallelisierung) verwendet.

### 2.2.1 Spielfeld und Spielzug

Listing 1: Struct Move

```
typedef struct {
        int fx; /**< x Coordinate of parent location. */
        int fy; /**< y Coordinate of parent location. */
        int tx; /**< x Coordinate of target location. */
        int ty; /**< y Coordinate of target location. */
} Move;
```

Listing 2: Struct Board

```
typedef struct {
        int pegs;     /**< Amount of pegs on the board.   */
        int xLength; /**< Length of the board's x-axis.  */
        int yLength; /**< Length of the board's y-axis.  */
        int **board; /**< A pointer to the actual board. */
} Board;
```

Da die Operationen, die mit dem Bewegen von Steinen auf dem Spielfeld zusammenhängen, mit großem Abstand am häufigsten ausgeführt werden war es unbedingt nötig, dass diese äußerst effizient durchgeführt werden. Zunächst wird das Spielfeld in einen Struct `Board` (siehe Listing 2) geladen, und anhand dessen werden die validen Spielzüge erstellt, die ebenso in einem Struct `Move` (siehe Listing 1) gespeichert werden. Das Spielfeld ist in dieser Form eine Matrix aus `integer` Werten. Eine Matrix ist aus mehreren Gründen nicht als effiziente Datenstruktur für ein Solitär-Spielfeld geeignet. Zum einen deckt sie Positionen ab, die nicht für das Spiel verwendet werden und zum anderen ist für die Information, ob an einer Stelle ein Stein steht oder nicht, lediglich 1 Bit benötigt. Da das Spielfeld 33 Positionen hat werden 33 Bits benötigt um eine Konstellation eindeutig zu codieren. Das Spielfeld so, in einem 64 Bit integer, codiert sähe am Anfang und am Ende so aus:

- START `111111111111111101111111111111111 000..`
- ENDE `000000000000000010000000000000000 000..`

Um zu dem Spielfeld in Bit-Repräsentation passende Spielzüge zu erstellen, war es nötig zunächst alle validen Spielzüge in Struct-Form zu finden. Es ist leicht anhand der Matrix festzustellen, welche Spielzüge innerhalb des Spielfeldes stattfinden und so vorgesehen sind. Daher werden zunächst alle Spielzüge in dem einfachen Format gespeichert. Danach werden für jeden so gefunden Spielzug zwei Bitmasken erstellt, die nötig sind um einen Spielzug auf dem Spielfeld in Bit-Repräsentation durchzuführen.

**Spielzüge** Die Spielzüge bestehen aus einer Check- und einer Set-Maske um mit einfachen logischen Operationen prüfen zu können, ob ein Spielzug möglich ist und ihn dann auszuführen. Ein Beispiel illustriert dies:

- `Check: 0000001100000000000000000000000`
- `Set:   0000001110000000000000000000000`
- `Board: 1100011100010011000101111101001010`
- `Board & Set == Check?`
  - Ja: Zug möglich (Zug ausführen: `Board = Board XOR Set`)
  - Nein: Zug nicht möglich

### 2.2.2 Algorithmus

Listing 3: Rekursive Tiefensuche

```
/* Fuer jeden moeglichen Spielzug */
for (int i = 0; i < moveCount; i++) {
        /* Wenn der Spielzug moeglich ist     */
        /* und noch nicht ausgefuehrt wurde */
        if (!checkMove(moves[i]) && !(moves[i] == history[draw])) {
                /* Fuehre den Spielzug durch */
                doMove(moves[i], board);
                /* Merke den Spielzug */
                history[draw] = moves[i];
                /* Rufe die Funktion neu auf und */
                /* erhoehe den Spielzug-Zaehler  */
                if (bruteForce(draw + 1) {
                        return 1;
                }
                /* Der bestrittene Pfad ist keine Loesung          */
                /* Mache den Zug rueckgaengig (Pfad + Spielfeld) */
                undoMove(history[draw], board);
                history[draw] = 0;
        }
}
return 0
```

**Naiver Ansatz** Ein erster Ansatz um Lösungen zu finden war eine Tiefensuche ohne weitergehende Optimierungen – also ein („Bruteforce“) Verfahren (siehe Listing 3). Das größte Problem bei diesem Verfahren ist, dass Spielfelder mehrfach auftreten, diese Information aber nicht genutzt wird. Da die

Anzahl der Lösungen gezählt werden soll, ist es relevant, *wie* man zu einem Spielfeld gelangt ist. Es ist ohne weiteres möglich zu einem beliebigen Spielfeld auf verschiedenen Wegen zu gelangen. Dabei muss nur 1 Zug unterschiedlich sein und ein neuer Lösungsweg ist entstanden. Der naive Ansatz für den Algorithmus sah so aus, dass alle Zweige des Suchbaumes, die valide Spielsequenzen repräsentieren, abgesucht und die dabei gefundenen Lösungen gezählt werden. Auf diese Art wird jedes Spielfeld unzählige male gelöst, was sich in einer unmöglich langen Ausführzeit wiederspiegelt.

**Dynamische Programmierung** Da der erste Ansatz (wie erwartet) nicht praktikabel war, musste ein neuer Algorithmus entwickelt werden. Mit Hilfe von Julian Kunkel wurde eine Breitensuche entwickelt, die das Prinzip der dynamischen Programmierung anwendet. Bei diesem Algorithmus wird jedes Spielfeld nur einmal gelöst. Wenn beim Setzen der Steine ein neues Spielfeld entsteht, wird überprüft ob dieses bereits an anderer Stelle auftrat. Ist dies der Fall, wird das Spielfeld nicht ein zweites Mal gespeichert, sondern lediglich der Zähler vom zuerst gefundenen Spielfeld erhöht. Die Idee für die Parallelisierung des Algorithmus bestand darin, den einzelnen Prozessen Teilmengen der vorhandenen Spielfelder zuzuweisen. Auf diese Weise berechnet ein Prozess nur einen Teil des Problems. Nach der letzten Iteration werden die Lösungen dieser Teilprobleme zur Gesamtlösung zusammengesetzt. Diese Vorgehensweise setzt voraus, dass sich die Prozesse nach jeder Iteration synchronisieren, sonst kommt es vor, dass Spielfelder redundant berechnet werden. In einer Iteration werden immer alle Spielfelder, die dem Prozess zugeordnet sind, betrachtet und für jedes dieser Spielfelder werden alle möglichen Spielzüge durchgeführt. In der ersten Iteration haben alle Spielfelder noch 32 Steine, danach nur noch 31. In der 31. Iteration haben die Spielfelder noch 2 Steine, danach nur noch 1. Die Aufteilung der Spielfelder auf die vorhandenen Prozesse (vgl. Abschnitt 2.2.3), die Verwaltung der Spielfelder und Lösungen (vgl. Abschnitt 2.2.4) sowie die Kommunikation (vgl. Abschnitt 2.2.5) werden im nachfolgenden besprochen.

### 2.2.3 Zuordnung Spielfeld zu Prozess

Die Zuordnung von Spielfeld zu Prozess ist zur Zeit Ursache der Lastungleichheit, da die Aufteilung statisch ist (vgl. Abschnitt 2.1.6). Schematisch dargestellt läuft die Zuordnung so ab:

- Prozesse: 16
- `Rang:    1100`
- `Board 1: 11000101000100110001000000100101 0`
- `Board 2: 1100011101010011100101111010110 10`
- `Board 3: 1101011000010011000101100011011 10`

Es werden die ersten Bits des Spielfeldes mit dem Rang des jeweiligen Prozesses verglichen. Stimmen sie überein wird das Spielfeld von diesem Prozess bearbeitet (Board 1 und Board 2), ist das nicht der Fall von einem anderen (Board 3).

### 2.2.4 Verwaltung der Spielfelder und Lösungen

Da während jeder Iteration ständig neue Spielfelder entstehen und für jedes geprüft werden muss, ob es schon einmal auftrat oder nicht, war eine Datenstruktur benötigt, die dieses effizient ermöglicht. Die Wahl fiel auf auf die Hashtable Implementation der GLib-2.0. Ein Eintrag im Hashtable besteht aus einem Schlüssel sowie einem dem Schlüssel zugeordnetem Wert. Der Schlüssel ist in diesem Fall jeweils ein Spielfeld und der dazugehörige Wert gibt an, wie oft dieses Spielfeld auftrat. Am Ende gibt es nur noch einen Eintrag, in dem Hashtable des Prozesses, dem das finale Spielfeld zugeordnet ist, der angibt wie viele Lösungen es insgesamt gibt. Erwähnenswert ist, dass zu einem Hashtable immer eine Hashfunktion gehört. In diesem Fall ist dies nicht nötig (streng genommen schon, es handelt sich aber um einen "direct-hash", d.h. wenn ein Schlüssel eingegeben wird, ist der Schlüssel automatisch auch der dazugehörige Hash), da ein Spielfeld in unserem Fall ein eindeutiger Hash ist.

### 2.2.5 Kommunikation

Der Algorithmus sieht vor, dass nach jeder Iteration eine Synchronisation zwischen den Prozessen stattfindet, damit Spielfelder nicht redundant berechnet werden. Am Ende einer Iteration hat jeder Prozess einen Hashtable, der alle Spielfelder enthält, die im Zuge der Iteration entstanden sind. Dazu gehören auch Spielfelder, die durch die Veränderung des Spielfeldes nicht mehr diesem Prozess zugeordnet sind. Daher müssen diese Spielfelder, die sich in dem Hashtable `nextBoards` befinden neu sortiert werden. Hierfür erstellt jeder Prozess für jeden Prozess (auch sich selbst) einen Hashtable, in den er die dem jeweiligen Prozess zugeordneten Spielfelder einfügt. Theoretisch könnten diese so verschickt werden, da MPI jedoch keine GLib Hashtables kennt müssen die Inhalte dieser Hashtables zunächst in einem Zwischenschritt in Arrays umkopiert werden (siehe Listing 4).

Listing 4: Umkopieren in Arrays

```
/* iterate over hashtable and copy key & value into array */
g_hash_table_iter_init(&iter, nextBoards);
while (g_hash_table_iter_next(&iter, &board, &boardSolutions)) {
        rank = assignBoardToProcess((bType) board);
        arrays[0][rank][count[rank]] = (bType) board;
        arrays[2][rank][count[rank]] = (bType) boardSolutions;
        count[rank]++;
}
```

Gleichzeitig muss auch Speicherplatz für die zu empfangenden Arrays bereitgestellt werden. Es ergeben sich 4 Arrays, die von jedem Prozess für jeden Prozess erstellt werden müssen. Jeweils Schlüssel und Wert zum Senden und Empfangen.

- **`arrays[0][Anzahl Prozesse][Elemente: Senden]`** speichert für alle Prozesse die zu verschickenden Spielfelder.
- **`arrays[1][Anzahl Prozesse][Elemente: Empfangen]`** speichert für alle Prozesse die zu empfangenden Spielfelder.
- **`arrays[2][Anzahl Prozesse][Elemente: Senden]`** speichert für alle Prozesse die zu verschickenden Lösungszähler.
- **`arrays[3][Anzahl Prozesse][Elemente: Empfangen]`** speichert für alle Prozesse die zu empfangenden Lösungszähler.

Diese Arrays werden anschließend mittels `MPI_Isend` und `MPI_Irecv` verschickt.

Listing 5: Verschicken der Arrays

```
/* exchange arrays */
for (int i = nprocs; i--;) {
        MPI_Isend(arrays[0][i], countSend[i], MPI_LONG, i,
                                    BOARDS, MCW, &requests[i]);
        MPI_Isend(arrays[2][i], countSend[i], MPI_LONG, i,
                        SOLUTIONS, MCW, &requests[nprocs+i]);
        MPI_Irecv(arrays[1][i], countRecv[i], MPI_LONG, i,
                        BOARDS, MCW, &requests[(2*nprocs)+i]);
        MPI_Irecv(arrays[3][i], countRecv[i], MPI_LONG, i,
                        SOLUTIONS, MCW, &requests[(3*nprocs)+i]);
}
MPI_Waitall(nprocs * 4,requests,MPI_STATUSES_IGNORE);
```

Nachdem die Arrays verschickt wurden, kann ein neuer Hashtable konstruiert werden, der alle diesem Prozess zugeordneten Spielfelder enthält. Dieser Hashtable (`currBoards`) wird dann verwendet um in der nächsten Iteration wieder `nextBoards` zu füllen.

Listing 6: Konstruktion von neuem Hashtable

```
/* Einen neuen Hashtable erstellen */
GHashTable *currBoards = g_hash_table_new(direct_hash, direct_equal);
/* Fuer jeden Prozess */
for (int i = nprocs; i--;) {
        /* Ueber die Anzahl der Elemente iterieren */
        for (int j = countRecv[i]; j--;) {
                /* Schauen ob Spielfeld im H-Table schon existiert */
                oldValue = g_hash_table_lookup(currBoards,
                                        (gpointer) arrays[1][i][j]);
                /* Loesungszaehler addieren */
                newValue = ((bType) oldValue) + arrays[3][i][j];
                /* Loesungszaehler ersetzten oder neu einfuegen */
                g_hash_table_replace(currBoards,
                                        (gpointer) arrays[1][i][j],
                                        (gpointer) newValue);
        }
}
```

### 2.3 Ergebnisse

Mit dem neuen Algorithmus war es möglich alle Lösungen für das Spiel zu finden. Validiert wurde das Ergebnis mit einem kleineren Spielfeld, welches deutlich weniger Lösungen liefert und außerdem nur 12 Positionen hat. Ferner wurde dieses Problem schon des öfteren gelöst und es gibt deshalb viele Quellen mit Hilfe derer das Ergebnis validiert wurde [1] [2].

#### 2.3.1 Laufzeiten

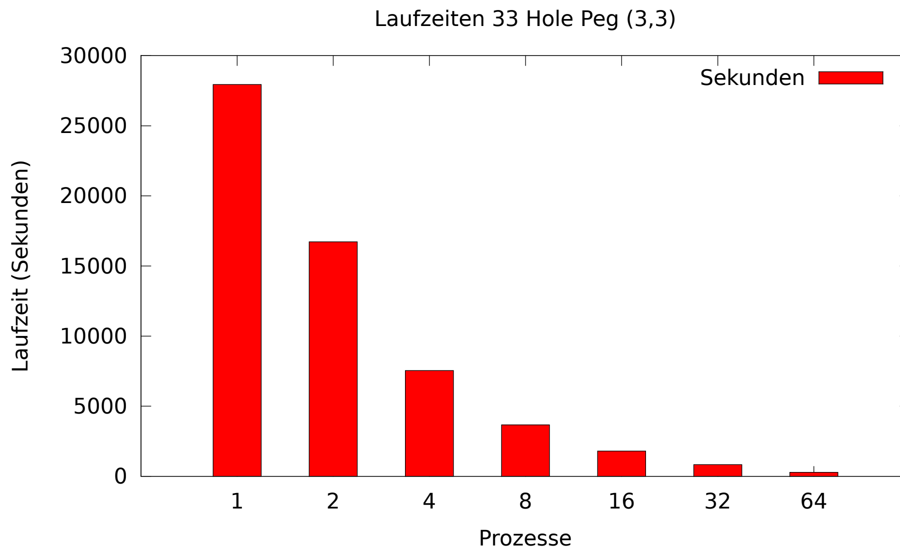


Abbildung 2: Laufzeiten in Sekunden je Anzahl Prozesse

Mit einem Prozess rechnet das Programm circa 8 Stunden am Standard-Solitär mit der mittigen Startposition. 64 Prozesse (auf 8 Knoten – 8/12 CPUs pro Knoten werden verwendet) lösen das Spiel in circa 5 Minuten.

### 2.3.2 Lösungen / Sekunde

Wie viele Lösungen pro Sekunde berechnet werden ist sehr interessant wenn man die Werte mit dem ersten Lösungsansatz (rekursive Tiefensuche) vergleicht. Lässt man diese einige Stunden laufen und errechnet, wie viele Lösungen durchschnittlich pro Sekunde gefunden werden, wird klar wie viel es ausmacht, dass in dem neuen Ansatz Spielfelder nicht länger mehrfach gelöst werden. Bei der Tiefensuche werden durchschnittlich 361 Lösungen pro Sekunde gefunden (nach 1 Stunde Laufzeit – nach 3 Stunden ist der Durchschnitt noch geringer, es variiert also) wohingegen 1 Prozess mit dem neuen Ansatz bereits 1462478419473 Lösungen pro Sekunde findet. Das heißt, bei dieser Geschwindigkeit (361 „LpS“) würde das Programm 31441710557 Stunden bzw. ca. 3589236 Jahre benötigen, um alle Lösungen zu finden!

| Prozesse | Lösungen / Sekunde |
|---|---|
| 1 (Tiefensuche) | 361 |
| 1 | 1462478419473 |
| 2 | 2442563634412 |
| 4 | 5415725253821 |
| 8 | 11149153353364 |
| 16 | 22688310405374 |
| 32 | 48994780623597 |
| 64 | 143374200140631 |

Abbildung 3: Lösungen pro Sekunde

### 2.3.3 Speedup

Der Speedup ist superlinear, wobei die Kurve mit mehr als 64 Prozessen abflachen würde. Der große Sprung zwischen 16 und 32 (bzw. 64 Prozessen) lässt sich damit erklären, dass die Hashtables bei so vielen Prozessen bei jedem Prozess klein genug sind um im Cache der CPU zu bleiben.

### 2.3.4 Hashtablesize

Da die Verteilung der Hashtables statisch verläuft und somit der Grund für die Lastungleichheit ist, lässt sich letztere über die Hashtables sehr gut visualisieren. Da, wie bereits erwähnt (vgl. Abschnitt 2.1.6), die Lastungleichheit auch dadurch entsteht, dass die Spielfelder unterschiedlich ergiebig sind, ist

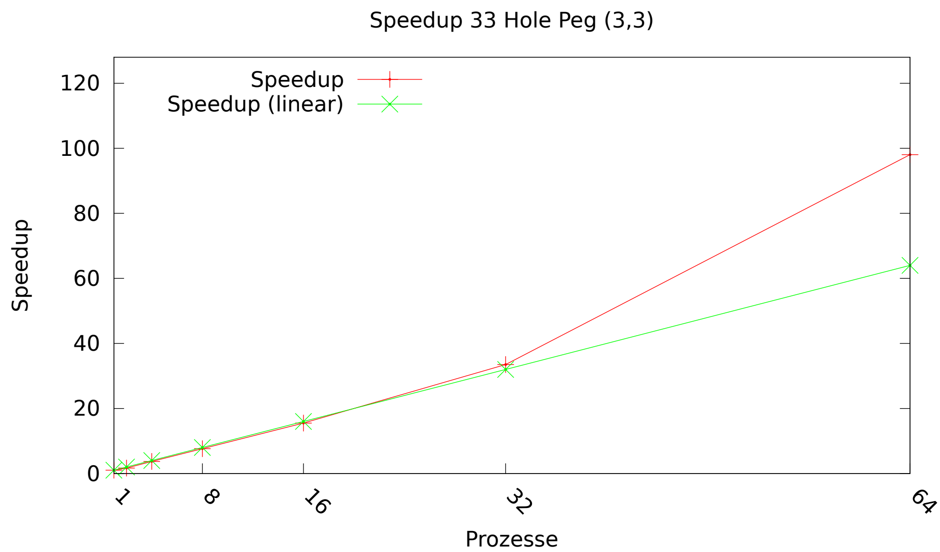


Abbildung 4: Speedup von Solitaire

disese Visualisierung keine 100%ige Übereinstimmung mit der *echten* Lastungleichheit.
Anhand der Anzahl der Elemente in den Hashtables lässt sich der maximale Speicherverbrauch (circa) pro Iteration berechnen. Mitsamt des allozierten Sende- und Empfangs-Puffers für den Austausch der Arrays beträgt er pro Element $((2 \cdot 8) \cdot 3)$ Byte.

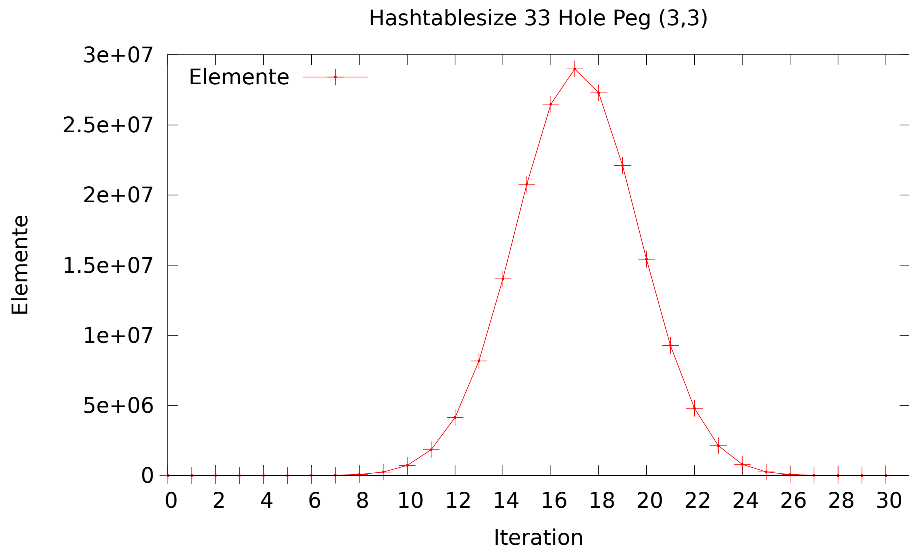


Abbildung 5: Anzahl der Spielfelder zu jeder Iteration

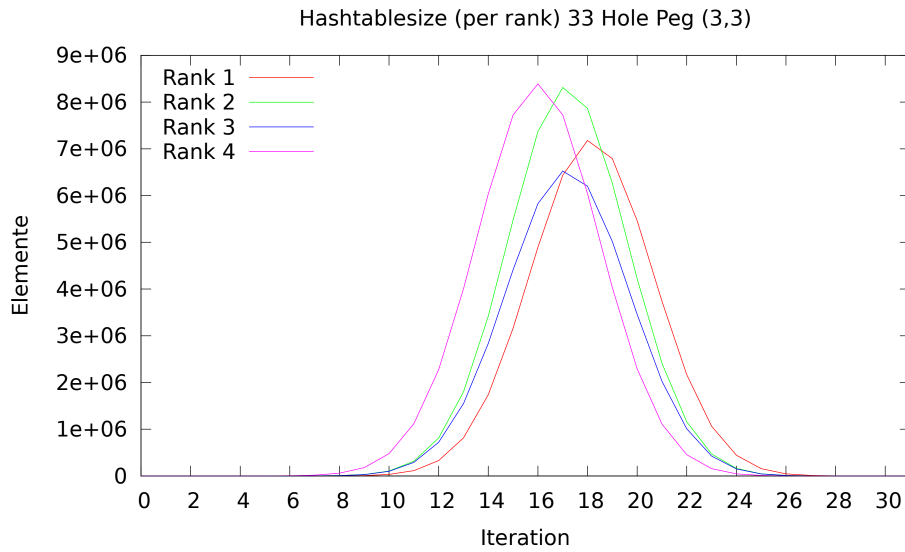


Abbildung 6: Verteilung der Spielfelder – 4 Prozesse

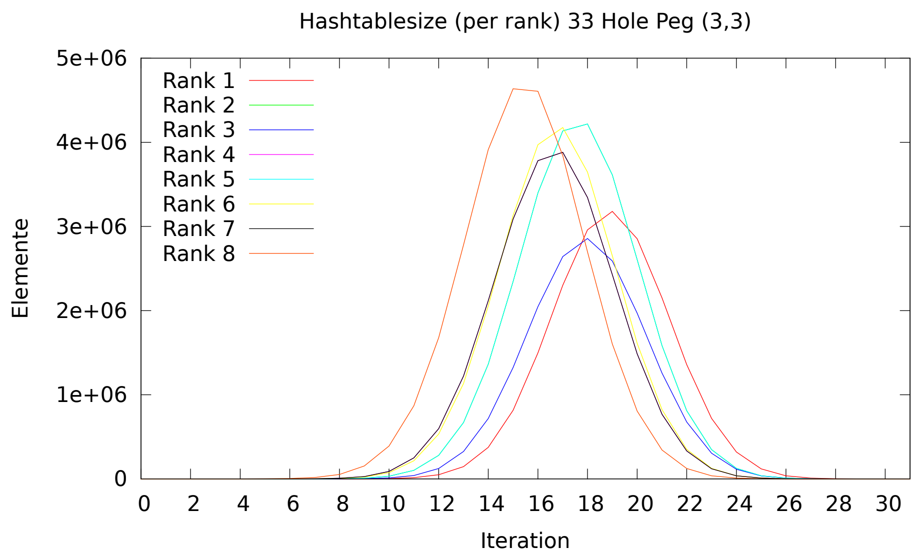


Abbildung 7: Verteilung der Spielfelder – 8 Prozesse

### 2.3.5 Tracing

Wie erwartet spiegelt sich im Trace[2] des Programmes die Lastungleichheit wieder. Ein Trace mit 8 Prozessen (siehe Abbildung 8) zeigt deutlich (erkennbar an den lilafarbenen Balken, welche visualisieren wie lange ein Prozess aktiv in `MPI_Waitall` gewartet hat), dass einige Prozesse mehr und andere weniger zu tun hatten. Es ist auch eine deutliche Korrelation zwischen dem Trace und der Verteilung der Hashtables (siehe Abbildung 7) erkennbar

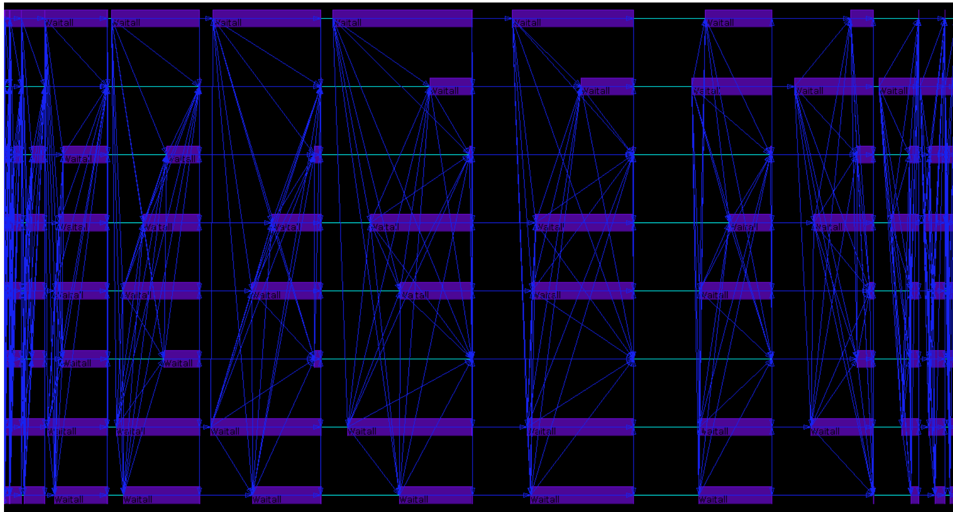

Abbildung 8: 33 Hole Peg (3,3) mit 8 Prozessen (1 pro Knoten)

[2] HDTrace – http://redmine.wr.informatik.uni-hamburg.de/projects/piosimhd

# 3 Zusammenfassung

Solitär zu lösen scheint auf den ersten Blick kein allzu schweres Problem zu sein, aber alle Lösungen für ein Spiel zu finden ist es auf jeden Fall. Das Problem ist sehr komplex und eignet sich hervorragend um Konzepte des parallelen Programmierens anzuwenden.
Dieses Praktikum hat meine Erwartungen voll und ganz erfüllt. Die Modellierung, Implementierung und Parallelisierung des Problems deckt viele wichtige Bereiche der Informatik hab, weswegen das Praktikum über den gesamten Zeitraum sehr lehrreich war. Nicht nur das Programmieren in C, sondern auch die Möglichkeit Erfahrungen in der Verwendung eines Clusters[3] zu sammeln sind für das weitere Studium äußerst wertvoll. Sehr hilfreich war auch die Möglichkeit aktuelle Software aus dem Bereich Versionsverwaltung zu verwenden[4].
Das Praktikum war sehr umfangreich und anspruchsvoll, aber wann immer ich Probleme hatte konnte ich meinen Betreuer Julian Kunkel fragen, der mir jedes Mal weiterhelfen konnte.

[3]http://wr.informatik.uni-hamburg.de/teaching/ressourcen/cluster
[4]http://stud.wr.informatik.uni-hamburg.de/projects/10-solitaire (Redmine, Git)

## Literatur

[1] Sunith Bandaru. *Durango Bill's 33 Hole Peg Solitaire*. 2011. URL: http://home.iitk.ac.in/~sunithb/reports.html.

[2] Bill Butler. *Durango Bill's 33 Hole Peg Solitaire*. 2011. URL: http://www.durangobill.com/Peg33.html.