Praktikum Paralleles Programmieren: Suchbaumproblem Drop7

Jakob Lüttgau
9luettga@informatik.uni-hamburg.de

29. November 2012

Zusammenfassung

Beschreibt den Lösungsversuch des Suchbaumproblems zum Spiel Drop7 auf einem parallen Mehrkernsystems unter Zuhilfenahme von MPI.

Inhaltsverzeichnis

1 Problembeschreibung

1.1 Drop7

„Drop7“ ist ein tetrisartiges Spiel, das auf Android und IOS Geräten gespielt werden kann. Auf einem 7x7 Gitter kann der Spieler dann ähnlich wie bei 4-Gewinnt Scheiben von oben in das Spielfeld einfügen. Mit etwas Geschick können so Kettenreaktionen ausgelöst werden, für die der Spieler Punkte erhält.

Jede Scheibe hat einen Wert von 1 bis 7, dieser Wert kann sichtbar oder verdeckt sein.

Die Scheiben lassen sich außerdem horizontalen und vertikalen Gruppen verschiedener Größe zuordnen. Stimmt der Wert einer unverdeckten Scheibe mit der Größe einer Gruppe, der die Scheibe angehört, überein, so platzt die Scheibe und beschädigt dabei umliegende noch verdeckte Scheiben.

In Regelmäßigen Abständen rückt von Unten eine Zeile verdeckter Scheiben ins Spielfeld und erhöht so den Druck auf den Spieler.

Der Spieler verliert wenn das Spielfeld voll ist, oder wenn Scheiben beim Level-Wechsel aus dem oberen Spielfeldrand herauswachsen.

1.2 „Sequence Mode“ – Suchbaumproblem

Neben Zufallsspielen und verschiedenen Schwierigkeitsstufen gibt es auch einen Spielmodus, bei dem die Sequenz der Scheiben fest vorgegeben ist. Der Spieler kann sich also die Spielfolge merken und eine Strategie entwickeln um einen möglichst hohen Punktestand zu erreichen.

Aus der Android-Anwendung lässt sich z.B. mit Hilfe von verschiedenen Werkzeugen[1][2][3][4] die Sequenz herausfinden. Wie sich herausstellt hat sie die Länge 3751.

Mit jedem Zug tun sich dem Spieler nun bis zu 7 neue Wege auf, es handelt sich also um ein Suchbaumproblem bei dem sich die möglichen Spielverläufe

[1] https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup
[2] http://code.google.com/p/android-apktool/downloads/list
[3] http://code.google.com/p/dex2jar/
[4] http://java.decompiler.free.fr/?q=jdeclipse

im schlimmsten Fall wie folgt ergeben:

$$\text{Möglichkeiten} < 7^{3751} \approx 9.178000847834 \times 10^{3169} \quad (1)$$

Viele dieser Wege müssen aber nicht so weit berechnet werden, da ein Spiel z.B. schon nach wenigen hundert Zügen verloren wird. So gibt es vielleicht gar kein Spiel das wirklich 3751 Züge lang ist.

2 Umsetzung

Die Drop7 Spiellogik soll in C implementiert werden, zunächst als Sequenzielles Programm. In einem zweitem Schritt soll dann ein Programm entwickelt werden, das auf einem Clustercomputer über mehrere Prozessoren verteilt eine Wartschlange abarbeiten kann und gegebenenfalls Snapshots im Dateisystem ablegen kann um die Berechnung zu einem späteren Zeitpunkt fortzusetzen. Auch dieses Programm soll in C geschrieben sein und nutzt die Bibliothek MPI zur Kommunikation zwischen den Rechner-Knoten.

2.1 Drop7

2.1.1 Spielstand festhalten

Um die Arbeit später leichter zwischen den Prozessoren verteilen zu können und weil es die Programmierung vermutlich erleichtert, soll ein Datentyp der den Spielstand festhält eingeführt werden. In C bieten sich dafür Structs an, die das Spielfeld, die gespielte Sequenz und einige andere Spielinformationen speichern:

```
#define D7_STATUS_CALCULATE 0
#define D7_STATUS_REQUIRE_DISC 1
#define D7_STATUS_GAME_OVER 3

typedef struct d7 {
        int m[7][7];          // Spielfeld
        int sequence[3750]; // Spielfolge
        int score;
        int level;
        int nextleveldiscs;
        int turn;
        int discs;
        int status;           // vgl. D7_STATUS_*
```

```
        int depth;
} d7;
```

2.1.2 Spielschritt-Berechnung

Dieser Teil implementiert analog zum verhalten von Drop7 einige Funktionen mit denen sich zu einem Spielstand D7 der Nachfolgezustand D7', in Abhängigkeit einer zuvor platzierten Scheibe, berechnen lässt.

Dieser Teil verdient besonders viel Zeit zur Optimierung, denn Idealerweise verbringt das spätere Programm möglichst viel Zeit damit, die Spielverläufe zu berechnen. An dieser Stelle soll der Algorithmus allerdings nur grob vorgestellt werden.

```
typedef struct _d7_impact {
        int m[7][7];
        int poppers;
} d7_impact;

void d7_call( const char* format, ... );

d7 dropdisc(d7, int, int);

d7 calcturn(d7);

d7 boardclear (d7 game);

d7_impact poppers_col(d7);
d7_impact poppers_row(d7);

d7_impact merge_impact(d7_impact, d7_impact);

d7 apply_impact(d7, d7_impact, int depth);

d7 gravity(d7);

d7 levelup(d7, int* row);
```

Auf dem Weg von D7 nach D7' verändert sich das Spielfeld und platzende Scheiben werden je nach Kettenreaktionslänge mit Punkten belohnt.

Um zu berechnen ob eine Scheibe platzt muss in den Zeilen und Spalten nach Gruppen gesucht und die Gruppengröße festgehalten gehalten werden.

Dazu stehen in entwickelten C Implementationen die Funktionen poppers_row() und poppers_col() zur Verfügung, die Ergebnisse der beiden Funktionen müssen dann noch zu einer Veränderungsmatrix zusammengeführt werden.

Diese Veränderungsmatrix lässt sich dann wie bei Matrizen üblich einfach auf die Spielfeldmatrix addieren.

Hat sich das Spielfeld beruhigt, erhalten wir eine Nullmatrix als Veränderungsmatrix. Andernfalls ruft sich die Spielschritt Berechnung Rekursiv erneut auf bis wir eine Nullmatrix erhalten.

2.2 MPI balancierte Warteschlange

Ein paar Annahmen zum Problem um Drop7 die aufs Design der Umsetzung Auswirkungen hatten:

- Großer Speicherbedarf, mehr als RAM verfügbar
- Berechnetes lässt sich schlecht wiederverwenden, da..
  - spezifisch auf Einzelfall, zumindest nach kurzer Zeit
  - und selbst wenn dann bloß noch ein zusätzliches Speicherproblem
- Sortiertes abarbeiten der Warteschlange verspricht kaum Vorteile
  - Wann würden denn überhaupt z.B. mehr Cachehits erzielt? Die Einzelfall-Hypothese deutet ja sowieso auf keine zuverlässigen Cachehits
- Zu komplexe Ansätze sind in der gegebenen Zeit nicht Umsetzbar

2.2.1 Warteschlange

Die Warteschlange muss dynamisch wachsen können, und sich im Fall von zu wenig Hauptspeicher auslagern lassen. Ein Tiefensuche-Algorithmus sollte dabei den Speicherbedarf langsamer ansteigen lassen als eine breit angelegte Suche.

Die Tiefensuche lässt sich außerdem mit geringem Aufwand implementieren. Etwa durch ein Array und einem Zähler der den Index des jüngsten und damit nächsten Elements speichert. Mit Push und Pop wird der Zähler dann inkrementiert bzw. dekrementiert.

Kompliziertere Datenstrukturen (etwa verkettete Listen, Suchbäume oder Hashtables) versprechen abgesehen vom Implementationsaufwand keine Vorteile.

2.2.2 Kommunikation

Zum einfachen Versenden von Spielständen bietet es sich an einen eigenen MPI-Datentyp zu registrieren. Das folgende Beispiel sollte das Vorgehen verdeutlichen:

```
#include <stddef.h>
#include <mpi.h>
#include "d7.h"

MPI_Datatype mpi_d7;

MPI_Datatype types[] = {
        MPI_INT, // m
        MPI_INT, // sequence
        MPI_INT, // score
};

int blocklengths[] = {
        7*7,  // m
        3750, // sequence
        1,    // score
};

MPI_Aint displacements[] = {
        offsetof(d7, m),
        offsetof(d7, sequence),
        offsetof(d7, score),
};

// init mpi
MPI_Init( &argc, &argv );
```

```
// create costum datatype
MPI_Type_create_struct(
                3,
                blocklengths, displacements, types,
                &mpi_d7
);

// ...

MPI_Send( &buf, 1, mpi_d7, rank, tag, MPI_COMM_WORLD );
```

Die Problemberechnung erfordert im Grunde keine Reihenfolge, dem entsprechend sollte die Kommunikation auch unkompliziert ausfallen. Nötig wird Kommunikation in zwei Fällen, zum einen wenn der Hauptspeicher nicht mehr ausreicht um die Warteschlange zu speichern und zum anderen wenn mehr Spiele verloren gehen als neue Spielverläufe entstehen, die Warteschlange also leer läuft.

MPI selbst unterstützt keine Interrupts, etwa für den Fall, dass ein Worker den Master oder einen anderen Worker mitteilen will das er bald keine Arbeit mehr haben wird.

Kommunikation muss also immer von beiden Partnern ausgehen. Durch nicht blockierendes Senden MPI_Isend / MPI_Irecv und MPI_Test lässt sich allerdings ein ganz ähnliches Verhalten nachempfinden.

In dem später gemessen Programm wurde das eben beschriebene Verfahren nicht verwendet da gelegentlich Verklemmungen entstanden sind, denen aufgrund eines gewissen Zeitdrucks nur in einem begrenzten Umfang nachgegangen wurde.

Realisiert wurde ein Periodischer Lastausgleich mit blockierendem Senden, nach jeweils z.B. 200.000 Arbeitszyklen übermitteln die Worker dem Master die Größe ihrer Wartschlange. Der Master erstellt eine Statistik und errechnet die durchschnittliche Warteschlangenlänge und verteilt bei Bedarf Arbeit um.

Das ist offensichtlich kein besonders cleveres Vorgehen, denn so entsteht eine Phase in der auf Kommunikation gewartet und nicht mehr gearbeitet wird. Bei großen Warteschlangen könnte die Kommunikationsphase dann sogar länger dauern als die Arbeitsphase. Verlängert man dann wieder die Ar-

beitsphase läuft man irgendwann Gefahr, dass ein Worker keine Arbeit mehr hat und dann deshalb wartet.

2.2.3 Persistenz

Nach Möglichkeit soll soviel Arbeit wie möglich im Hauptspeicher der Knoten abgelegt sein, erst wenn das Programm unterbrochen werden soll, oder wenn der Hauptspeicher nicht mehr ausreicht sollen Teile der Warteschlange im Dateisystemen abgelegt werden.

Praktisch wäre, wenn man die Structs der Spielstände einfach serialisiert in Dateien abspeichert. Wegen der knappen Zeit und weil wohl kaum eine robustere und portierbarere Lösung zustande kommt soll die Bibliothek TPL zum einfachen abspeichern von binären Daten benutzt werden.[5]

Eine Alternative zu TPL wäre EET[6] TPL schien auf den ersten Blick aber verständlicher dokumentiert.

```
#include "tpl.h"
// speichern
void dumpQueue() {
        tpl_node *tn;

        tn = tpl_map("A(S(i##i#iiiiiii))", &queue, 7, 7, 3750);

        for (int t = 0; t < queue_count; ++t) tpl_pack(tn,1);

        tpl_dump(tn, TPL_FILE, "dump.tpl");
        tpl_free(tn);
}

// laden
void loadQueueFromFile(struct dirent entry) {
        tpl_node *tn;
        d7 agame;

        tn = tpl_map("A(S(i##i#iiiiiii))", &agame, 7, 7, 3750);
        tpl_load(tn, TPL_FILE, "dump.tpl");
        while (tpl_unpack(tn,1) > 0)
                push(agame);
```

[5] http://tpl.sourceforge.net/

[6] http://www.enlightenment.org/p.php?p=about/efl/eet

```
        tpl_free(tn);
}
```

3 Leistungsanalyse

3.1 Geschwindigkeit

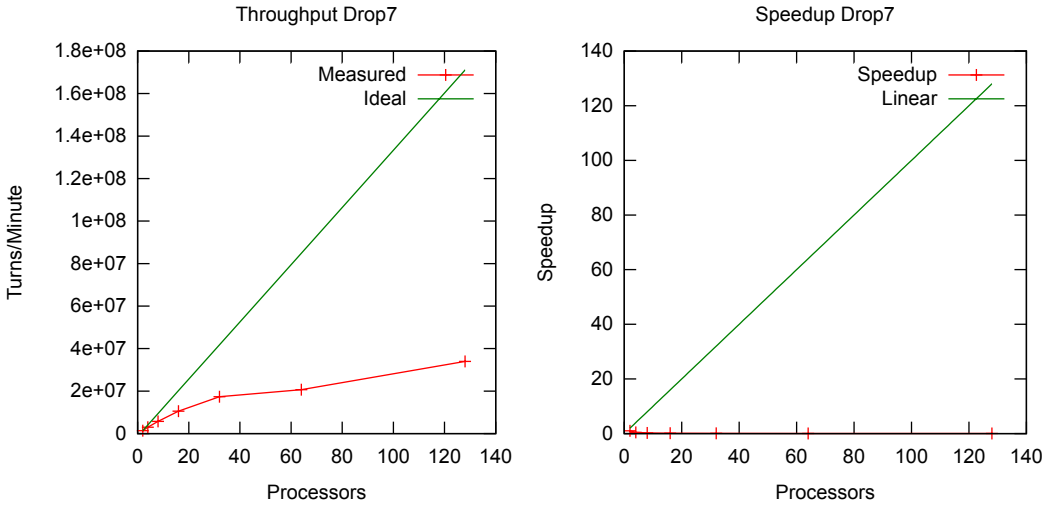

Abbildung 1: Leistungsmessung des Programms mit verschiedener Anzahl von Prozessoren. Leider mit kurzer Arbeitsphase gemessen.

Prozesse	Züge/Min.
2	1347179
4	3017708
8	5715230
16	10542499
32	17325913
64	20684082
128	33955173

3.2 Kommunikation

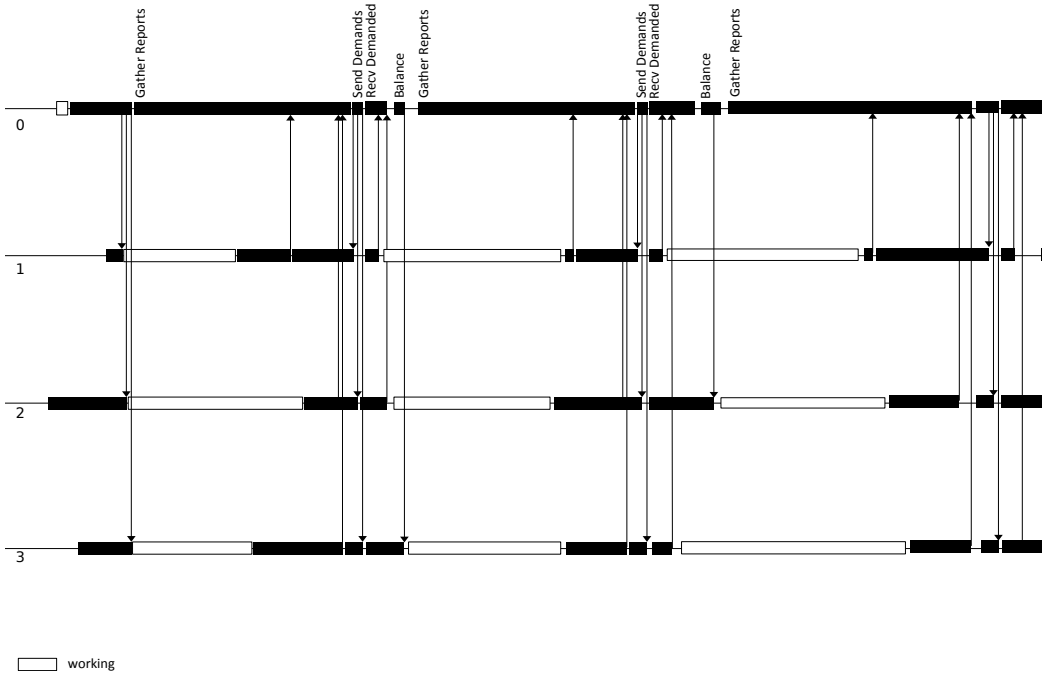

Abbildung 2: Schema der Kommunkiationsstruktur

Das gemessene Kommunikationsschema, ist letztendlich verantwortlich für den relativ flachen Speedup. Beim Lastausgleich entstehen unnötige Wartezeiten durch das blockierende Senden. Diesem Effekt lässt sich durch Wahl einer längeren Arbeitsperiode allerdings entgegenwirken so lange immer nur zu viel Arbeit verfügbar ist.

3.3 Speicher

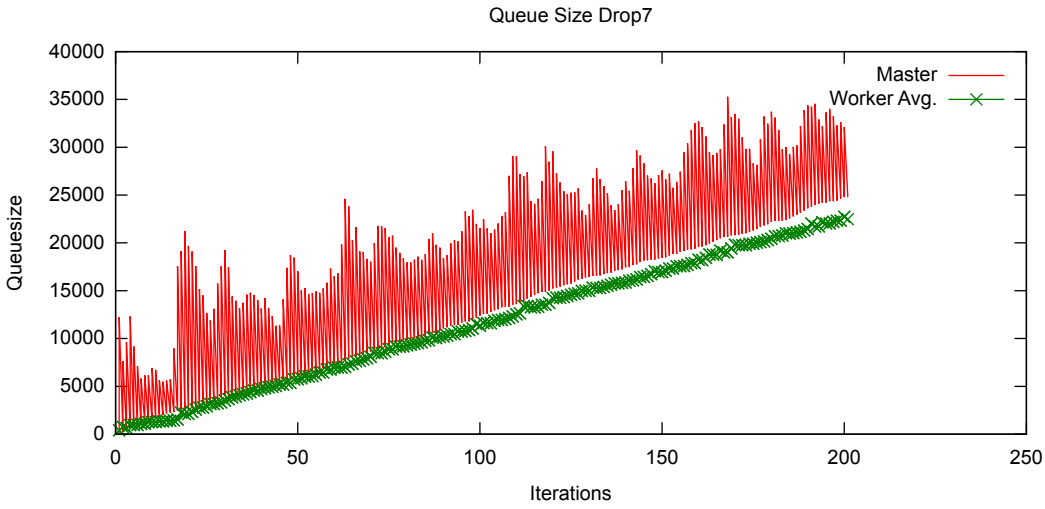

Abbildung 3: Hauptspeicherentwicklung über nach etwa 20 Mrd. gespielten Zügen

Die Speicherentwicklung ist soweit eigentlich nicht wirklich überraschend, schön wäre vielleicht gewesen wenn das Wachstum mit der Zeit leicht abgenommen hätte. Allerdings nicht aus Gründen der Leistungsbewertung sondern weil es die Problemgröße etwas entschärft.

4 Verbesserungsmöglichkeiten

Es gibt eine Reihe von Verbesserungsmöglichkeiten. Allen voran natürlich die Umsetzung des nicht blockierenden Sendens/Empfangens.

In der Master/Worker-Struktur könnte es sich auszahlen, dass jeder Knoten eigene Leistungsprognosen berechnet, und so z.B. die Arbeitsphase verlängert wenn lange gewartet werden musste.

Auch ist eine direktere Kommunikation ohne einen Master denkbar, bei der sich ein arbeitsloser Worker von einem benachbarten Worker neue Arbeit holt.

Ein recht vielversprechender Ansatz um Schnell zu Ergebnissen zu kommen könnte eine Spielheuristik oder AI sein, die zunächst vielversprechende Spiele weiterspielt.