Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

PRAKTIKUMSBERICHT: PARALLELE PROGRAMMIERUNG SS17

Vier Gewinnt

Vorgelegt von:	Timo Hahn
	Timo.Hahn@hotmail.de
Fakultät:	Fakultät für Informatik
Studiengang:	Wirtschaftsinformatik BSc.
Matrikelnummer:	6689942
Betreuer:	Dr. Julian Kunkel
Abgabedatum:	29.11.2017

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Das Spiel: Vier Gewinnt

Vier Gewinnt (engl. Captain`s Mistress) ist ein Strategiespiel für zwei Spieler mit dem Ziel als erster Spieler vier Spielsteine seiner Farbe in einer Reihe zu platzieren (horizontal, diagonal oder vertikal). Das Spiel wurde im Jahre 1974 veröffentlicht und findet auf einem Spielfeld mit sechs Reihen (waagerecht) und sieben Spalten (senkrecht) statt.

Abbildung 1: Spielfeld

Es gibt bei *Vier Gewinnt* zwei verschiedene Farben, in der Regel rote und gelbe Spielsteine. Jede Farbe gehört je einem Spieler, also gibt es einen gelben Spieler und einen roten Spieler. Spieler 1 beginnt mit dem Einwurf eines Spielsteins in eine beliebige der sieben Spalten. Als nächstes ist der zweite Spieler am Zug und tut es ebenso. In jede Spalte können maximal sechs Spielsteine eingeworfen werden bis diese voll ist. Die beiden Spieler wechseln sich mit ihren Zügen ab bis entweder ein Spieler vier seiner Steine in einer Reihe hat, hierbei zählt horizontal, vertikal oder diagonal, oder das Spielfeld voll ist ohne, dass ein Spieler die Siegesbedingung erreicht hat - dann endet das Spiel unentschieden.[1]

[1] vgl. Wikipedia: Vier Gewinnt

1.2 Der Minimax-Algorithmus

Der Minimax-Algorithmus dient zur Ermittlung des optimalen Spielzugs in einem endlichen Nullsummenspiel für zwei Spieler, was bedeutet, dass beide Spieler abwechselnd am Zug sind, bis das Spiel zu Ende ist. Eine Bedingung für die Nutzung dieses Algorithmus ist, dass *Vier Gewinnt* ein Spiel mit perfekter Information ist. Dies bedeutet, dass zu jedem Zeitpunkt einer Entscheidung, also dem Zug eines Spielers, ihm die vorangegangenen Entscheidungen bekannt sind. Der Minimax-Algorithmus wird beispielsweise auch in Schachcomputern verwendet, wodurch diese von durchschnittlichen Spielern nicht geschlagen werden können.

Der Minimax-Algorithmus benötigt einen Suchbaum, welcher alle möglichen Züge enthält und bewertet jeden einzelnen. Ausgehend von dieser Bewertung wird dann der nächste Zug getätigt, wobei eine hohe Bewertung gut für den ziehenden Spieler und eine niedrige Bewertung gut für den Gegenspieler ist.

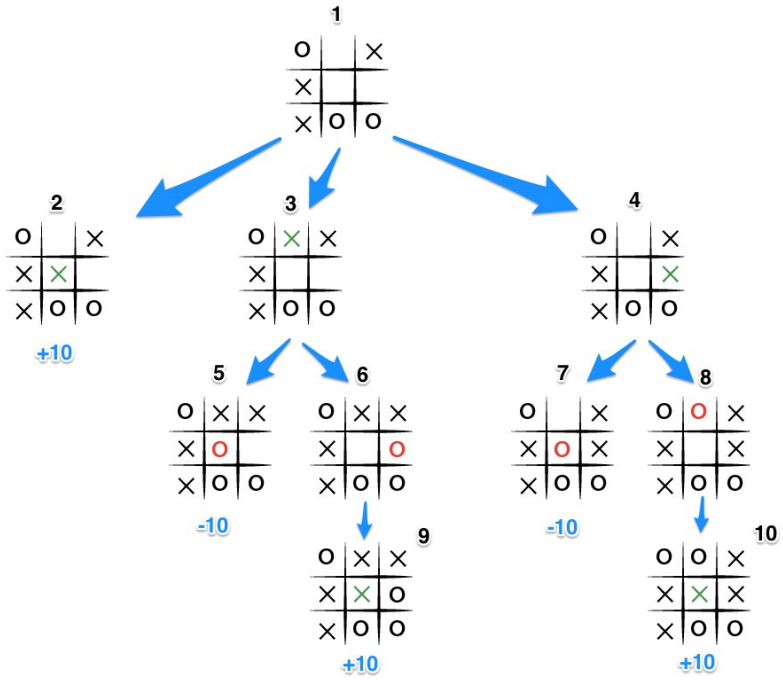

Abbildung 2: Darstellung des Minimax-Algorithmus

In Abbildung 2 wird jeder Siegesausgang vom X-Spieler (hier bei dem Spiel Tic-Tac-Toe) mit +10 bewertet und jede Niederlage mit -10. Mit dieser Information weiß Spieler X, dass er in Zug 2 gewinnt. Doch wenn er Zug 3 oder Zug 4 wählt und sein Gegner keine Fehler macht, also Siegessituationen immer wahrnimmt,

verliert. Für ein komplettes Spiel ist dieser Suchbaum sehr groß und besitzt bei *Vier Gewinnt* nach zwei vollendeten Spielzügen schon 56 Äste.[2]

1.3 Das Ziel

Das Ziel des Praktikumsprojektes ist es, das Spiel mit seinen Regeln in der Programmiersprache C umzusetzen. Dabei wird das Spiel nicht von Menschen manuell gespielt, sondern es wird der Minimax-Algorithmus eingesetzt und der Algorithmus spielt gegen sich selbst. Im nächsten Schritt wird das geschaffene Programm mit Hilfe von OpenMP parallelisiert. Da bei dem Ablauf des Programms keine menschlichen Verzögerungen involviert sind, kann man den Effekt der Parallelisierung anhand der Durchlaufzeit des Programms erkennen.

[2] vgl. Algorithms in a Nutshell, O`Reilly, 2016, S. 169 ff

2 Das serielle Programm

Das Grundgerüst des Programms bildet ein Array, welches das Spielfeld abbildet. Der Array hat die Anzahl an Elementen der Reihen multipliziert mit den Spalten plus eins. In der Annahme eines normalen Vier Gewinnt-Spiels bedeutet dies 7 * 6, also 42 Elemente. Hierbei repräsentiert das erste Elemente welche Zugnummer als nächstes kommt und die restlichen Elemente die geworfenen Spalten in aufsteigender Reihenfolge. Aus dieser Darstellung lässt sich eine zweite Darstellung ableiten, welche einer zweidimensionalen Darstellung des Spielfelds mit 0en, 1en und 2en, 0 für ein leeres Feld, 2 für Steine des ersten Spielers und 1 für Steine des zweiten Spielers, entspricht. Die Prüfung nach der Gewinnkondition, also vier Steinen in einer Reihe, erfolgt durch Iteration über das gesamte Spielfeld. Die Einbindung des Minimax-Algorithmus erfolgt über eine rekursive Funktion.

```
static void algorithm(field f, field g) {
    for(i=0;i<COLS;i++) {
        make_turn(i);
        if(no winner and board not full) {
            algorithm(f, g);
            undo_turn(g);
        }
        else if(there is a winner) {
            evaluate(g);
            undo_turn(g);
        }
        else if(no winner and board full) {
            evaluate(g);
            undo_turn(g);
        }
    }
}
```

Abbildung 3: Vereinfachte Darstellung Minimax

In Abbildung 3 ist die vereinfachte Implementierung des Algorithmus zu sehen. Die for-Schleife iteriert über jede einzelne Spalte und ruft so jeden möglichen Zug ab. Bei vollem Spielfeld oder einem vorhandenen Sieger, den beiden Abbruchbedingungen des Spiels, evaluiert das Programm eine Punktzahl für den Zug. Hierbei stehe eine 1 für einen Sieg, eine -1 für eine Niederlage und eine 0 für ein Unentschieden. Diese Punktzahlen werden pro Spalte miteinander abgeglichen, sodass auf oberster Ebene über einen Array die Punktzahlen der verschiedenen Spalten abgeglichen werden. In die Spalte mit der höchsten

Punktzahl wird nach Terminierung des Algorithmus der Stein tatsächlich hineingeworfen.

3 Die Parallelisierung

3.1 Die Implementierung

Für die Parallelisierung des Programms habe ich die Programmierschnittstelle OpenMP gewählt, welche für die Programmiersprachen C, C++ und Fortan existiert.[3] Der zu parallelisierende Bereich, also dort wo die meiste Arbeit anfällt, ist das Codestück welches auf Abbildung 3 zu sehen ist. Für meine Parallelisierung habe ich mich für eine Änderung der Codestruktur entschieden. Wenn ich über die for-Schleife parallelisiere, würden zu viele Daten hinsichtlich der Kommunikation zwischen den verschiedenen Threads anfallen, sodass kein Speedup mehr vorhanden wäre, sondern das Programm sogar länger brauchen würde. Deshalb habe ich die Algorithmus-Funktion geteilt.

```
static void algorithm_seq(field f, field g) {
    for(i=0;i<COLS;i++) {
        make_turn(i);
        if(no winner and board not full) {
            algorithm(f, g);
            undo_turn(g);
        }
        else if(there is a winner) {
            evaluate(g);
            undo_turn(g);
        }
        else if(no winner and board full) {
            evaluate(g);
            undo_turn(g);
        }
    }
}
```

Abbildung 4: Zusätliche Algorithmus-Funktion

Wie in Abbildung 4 zu sehen, entspricht die neue Funktion exakt der rekursiven Funktion, doch ruft sie statt sich selbst, die herkömmliche Funktion, aus Abbildung 3 bekannt, auf. Über diese for-Schleife parallelisiere ich nun den rechenintensiven Teil des Programms.

[3] vgl. OpenMP

```
static void algorithm_seq(field f, field g) {
    #pragma omp parallel
    {
    #pragma omp single
    {
    for(i=0;i<COLS;i++) {
        #pragma omp task firstprivate(g, c, i, p)
        {
        make_turn(i);
        if(no winner and board not full) {
            algorithm(f, g);
            undo_turn(g);
        }
        else if(there is a winner) {
            evaluate(g);
            undo_turn(g);
        }
        else if(no winner and board full) {
            evaluate(g);
            undo_turn(g);
        }}
    }}
    #pragma omp taskwait
    }
}
```

Abbildung 5: Parallelisierte Algorithmus-Funktion

In Abbildung 5 ist nun die parallelisierte Algorithmus-Funktion zu sehen. Die privaten Variablen sind die Kopie des Spielfeldes *g*, welches jeder Thread einzeln braucht, das umgewandelte Spielfeld *c*, die Variable *p* aus der rekursiven Funktion und *i* aus der for-Schleife. Die einzige weitere genutzte Variable ist f, doch diese kann den geteilten Status behalten, da in sie nicht geschrieben wird.[4]

Als erstes wird der parallele Teil eröffnet. Hier wird die Gruppe von Threads eröffnet. Damit jede Aufgabe nur einmal vergeben wird, wird danach ein Bereich deklariert, welcher von nur einem Thread ausgeführt wird. Innerhalb der for-Schleife wird dann eine Aufgabe, mit den zuvor erklärten Freigaben der Variablen, definiert. Am Ende wird noch ein Punkt gesetzt an dem gewartet wird bis alle Aufgaben erledigt sind.[5]

[4] vgl. Parallele Programmierung, Springer, 2007, S. 356f
[5] vgl. NERSC: Introduction to OpenMP tasks

3.2 Die Performance

Aufgrund der gewählten Parallelisierungsart ist die maximal verfügbare Anzahl an nützlichen Threads sehr gering. Sie entspricht der Anzahl vorhandenen Spalten in dem Spiel. Aufgrund der Komplexität des Spiels und der benötigten Zeit zur Terminierung des Algorithmus hat jegliches Testen auf einer beschränkten Feldgröße von 4 * 4 stattgefunden.

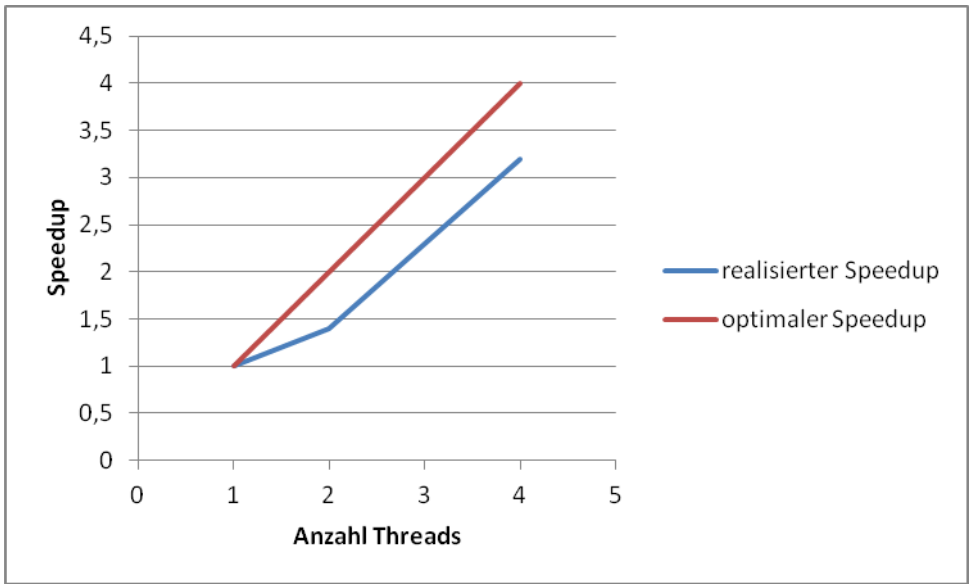

Abbildung 6: Speedupdiagramm

Den Werten des Diagramms liegen Durchschnittswerte aus jeweils drei Programmdurchläufen zu Grunde mit jeweils 1, 2 und 4 aktivierten Threads. Man erkennt, dass bei zwei genutzten Threads der Speedup nur minimal und die Diskrepanz zwischen den Werten des optimalen und realisierten Speedups am größten ist. Hier konnte nur ein Speedup von 1,4 im Durchschnitt erzielt werden. Bei vier Threads schneidet das Ergebnis wieder besser ab. Hier liegt der Speedup immerhin bei 3,2.

4 Kritische Reflexion

Das Modul war für mich sehr fordernd, da ich im Laufe meines Studiums nur die Softwareentwicklung-Grundkurse belegt habe und keine Erfahrung mit Datenstrukturen oder Algorithmen hatte. Im Laufe dieses Moduls lernte ich eine neue Programmiersprache kennen, sowie die Arbeit mit der Programmierschnittstelle OpenMP und das Debuggen.
Ich bin mit dem Umfang des erarbeiteten Wissens, der Erweiterung meines Wissens in der Breite und der Gesamtleistung im Projekt zufrieden. Mit dem Projektergebnis bin ich nicht zufrieden, da die Qualität der Umsetzung nicht meinen eigenen Ansprüchen genügt, doch aufgrund der fehlenden Kenntnisse keine bessere Leistung in einem angemessenen Zeitrahmen möglich war.
Die Parallelisierung ist für mich zufriedenstellend, doch terminiert das Programm nicht immer unter Einsatz mehrerer Threads. Das lässt auf ein fehlerhaftes serielles Programm schließen, da das parallele Programm, wenn es terminiert, in einem realistischen Speedup durchläuft.
Mein Zeitmanagement während der Durchführung des Projektes war suboptimal, da ich in Blöcken arbeiten musste aufgrund anderer universitärer Verpflichtungen und mich so immer wieder neu einarbeiten musste. Dies war zusätzlich aufwändig aufgrund des wenigen, wie eingangs erwähnten, gefestigten Wissens.
Rückblickend hätte ich lieber ein Thema mit einem Informatikstudenten in Partnerarbeit erledigt statt allein, um das Wissen aus Informatikkursen, wie z.B. Algorithmen und Datenstrukturen, verschiedene Lösungsansätze und einen kritischen Dialog im Team zu haben.

Literaturverzeichnis

Rauber, T., Rünger, G., Parallele Programmierung, 2. Auflage, Heidelberg: Springer Verlag, 2007

Heinemann, T., Pollice, G., Selkow, S., Algorithms in a Nutshell, 2nd edition, Sebastopol: O`Reilly, 2016

OpenMP, 2017, http://www.openmp.org/ (zuletzt geprüft 29.11.2017)

NERSC: Introduction to OpenMP tasks, 2017, http://www.nersc.gov/users/software/programming-models/openmp/openmp-tasking/openmp-task-intro/ (zuletzt geprüft 29.11.2017)

Wikipedia: Vier Gewinnt, 2017, https://de.wikipedia.org/wiki/Vier_gewinnt (zuletzt geprüft 29.11.2017)