Neugengo

— Praktikumsbericht —

Praktikum: Parallele Programmierung
Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Lennart Braun, Armin Schaare, Theresa Eimer
Betreuer:	Julian Kunkel

Hamburg, den 30.09.2015

Abstract

In diesem Projekt wurden eigens implementierte, künstliche neuronale Netze trainiert, das asiatische Brettspiel Go spielen zu können. Da das Training enorm vieler Kalkulationen bedarf, haben wir uns für die Umsetzung auf einem Rechencluster mit entsprechenden Parallelisierungsschemata auseinander gesetzt.

Inhaltsverzeichnis

1. Problemstellung

Neuronale Netze wurden in den letzten Jahren immer wieder für verschiedene Problemstellungen benutzt. Es wurde festgestellt, dass sie auch komplizierte Aufgaben lösen können, vorausgesetzt sie werden gut genug trainiert. Go zu spielen ist ein sehr komplexe Aufgabe, die wir unter anderem genau aus diesem Grund ausgewählt haben. Wegen der vielen Zugmöglichkeiten und dem mit 19×19 Punkten sehr großen Spielbrett, gibt es für Go noch keine Computer, die Menschen deutlich übertreffen oder sogar eine perfekte Strategie spielen können. Dieses Ziel wäre natürlich etwas hoch gegriffen, doch mit diesem Projekt wollten wir sehen, ob sich neuronale Netze überhaupt gegenseitig so trainieren können, dass sie bessere Ergebnisse erzielen.

Weiterhin liegt unser Fokus darauf, das Training auf einem Cluster zu parallelisieren. Dabei sollen die Berechnungen möglichst effizient auf ein System mit verteiltem Speicher aufgeteilt werden.

2. Über Go

Go ist ein asiatisches Brettspiel, das normalerweise auf Brettern von 19×19 oder 9×9 Schnittpunkten gespielt wird. Es gibt zwei Spieler, einer spielt schwarze, der andere weiße Steine. Nacheinander legt zuerst der schwarze, dann der weiße Spieler, je einen Stein auf einen Schnittpunkt. Ist ein Stein umzingelt, sind also alle seine vier direkt angrenzenden Schnittpunkte von gegnerischen Steinen besetzt, so ist er geschlagen und wird vom Brett genommen. Zusammen mit dem auf die selbe Weise umzingelten Gebiet am Ende des Spieles, bilden die geschlagenen Steine die Punktzahl der Spieler. Wer mehr Punkte hat gewinnt. Eine Sonderregel ist hier das Ko. Es ist möglich in eine Endlosschleife zu geraten, indem ein Stein immer wieder geschlagen und zurückgeschlagen wird. Damit das nicht passiert, darf in dieser Situation nicht sofort zurückgeschlagen werden, sondern erst einen Zug später um dem Gegner die Möglichkeit zu geben die Lücke zu schließen.

Um das Spiel etwas zu vereinfachen und schneller Ergebnisse zu sehen, arbeiten wir mit 9×9-Spielbrettern. Außerdem gibt es ein Zuglimit für die Spiele, damit es keine Endlosschleifen gibt, die trotz Ko auftreten können. Ansonsten haben wir uns aber bemüht die Go Regeln möglichst gut umzusetzen, damit die Spielstärke auch realistisch beurteilt werden kann.

3. Implementation

Zur Bearbeitung der Problemstellung, wurde die Programmiersprache C und überwiegend ein objektorientiertes Paradigma verwendet. Die Implementation besteht aus drei nahezu unabhängige Bereichen, dem Go, den neuronalen Netzen und dem Genetischen Algorithmus. Die benötigte Funktionalität wurde jeweils als Modul einer Bibliothek implementiert. Diese wurde mit dem Test Framework Check [1] während der Entwicklung regelmäßig auf Fehler untersucht.

Als Schnittstelle für den Benutzer wurden kleine Anwendungen geschrieben, die Algorithmen und Datenstrukturen aus der Bibliothek verwenden, um einzelne Aufgaben zu erfüllen.

3.1. Das neuronale Netz

Ein neuronales Netz besteht aus Neuronen, ähnlich einem menschlichen Gehirn. Die Neuronen sind in Schichten angeordnet, den Layern des Netzes. Das erste Layer ist dabei das Input Layer, das letzte das Output Layer. Zwischen den Layern gehen Kanten von jedem Neuron des einen zu jedem Neuron des anderen Layers. Diese Kanten haben Gewichte und pro Layer gibt es auch einen Bias, der zum Kantengewicht addiert wird. Ein Signal wird durch das Input Layer aufgenommen, durch die Layer weitergegeben und vom Output Layer wieder ausgegeben. Das Weiterleiten innerhalb des Netzes funktioniert über die Kanten. Der Output eines Neurons ist die Summe aller seiner Eingänge. Durch eine Sigmoidfunktion - hier $\frac{1}{1+e^{-x}}$ - werden diese Summen auf den Wertebereich $[-1; 1]$ reduziert, wobei die Funktion gleichzeitig die spätere Anwendung des Backpropagation Algorithmus ermöglicht. Der fertige Output des Neurons wird dann mit den jeweiligen Kantengewichten multipliziert und an die Neuronen des nächsten Layers weitergeleitet. Vor allem hier ist also sehr viel zu Berechnen.

Der Aufbau der Netze ist variabel, die Anzahl der Layer sowie die Anzahl der Neuronen pro Layer ist frei wählbar. Das Input Layer sollte allerdings für jeden Schnittpunkt des Eingabebrettes ein Neuron haben und das Output Layer entsprechend ein Neuron pro Schnittpunkt plus ein Neuron um Passen anzuzeigen. Die Ausgabe sind die Präferenzen der Netze, der Schnittpunkt mit dem höchsten Wert wird besetzt.

Damit die Netze zumindest die Regeln von vornherein erlernen, gibt es eine Methode für Supervised Learning; den Backpropagation Algorithmus. Für ihn muss bekannt sein, was das erwartete Ergebnis ist, weswegen die Netze so nur auf die Go Regeln und nicht auf die Taktik trainiert werden können. Der Algorithmus basiert darauf, dass der tatsächliche Ausgabewert mit dem erwarteten Wert verglichen wird und der so ermittelte Fehler zur Korrektur der Kantengewichte benutzt wird. Um die richtigen Korrekturwerte für die einzelnen Layer zu erhalten, muss die Ableitung der Fehlerfunktion durch die Ableitung des Kantengewichts berechnet werden. Dazu summiert man die Fehlersignale der vorhergehenden Schicht mal dem dazugehörigen Kantengewicht auf und multipliziert es mit der Ausgabe des Neurons. So wird nur der Teil des Fehlers, der tatsächlich von diesem Neuron verursacht wurde, korrigiert.

3.1.1. Dateiformat

Um die neuronalen Netze einfach wiederverwenden und übertragen zu können, wurden einfache Dateiformate entwickelt. In diesen lassen Mengen von Netzen entweder in menschenlesbarer Textform (Beispiel in Listing 1, Seite 19) oder in Binärform abspeichern.

3.2. Das Go-Modul

Die Regeln von Go sind hauptsächlich in dem Board Modul implementiert. Dieses stellt einen entsprechenden Datentypen (`board_t`) zur Verfügung, welcher das Spielfeld repräsentiert und die momentane Stellung in seinem Zustand speichert. Auf das Board können Steine platziert werden. Dabei wird überprüft, dass die Züge auch legal sind. Das Board legt die Gruppen in einer internen Datenstruktur ab und verbindet einen platzierten Stein mit den benachbarten Gruppen der selben Farbe. Mit den Gruppen ist jeweils die Anzahl der verbleibenden Freiheiten gespeichert. Nach dem Platzieren eines Steins wird überprüft, ob die benachbarten Gruppen des anderen Spielers ihre letzte Freiheit durch den Zug verloren haben und geschlagen werden müssen. Das Speichern der Gruppen ist wichtig, da die Freiheiten sonst bei jedem Zug neu berechnet werden müssten. Die Berechnung des Gebiets erfolgt mit einem Flood-Fill Algorithmus.

Das Klassendiagramm (Abbildung 1) gibt einen Überblick über den Zusammenhang der erstellten Datenstrukturen. Um die neuronalen Netze Go spielen zu lassen und dabei mit dem genetischen Algorithmus zu trainieren, sind weitere Datenstrukturen notwendig.

Der Spielablauf einer Partie Go wird durch das Game (`game_t`) modelliert. Dieses lässt zwei Player (`player_t`) auf einem Board gegeneinander spielen. Das Interface eines Player definiert eine Funktion, die, gegeben der Zustand eines Boards, den zu spielenden Zug zurückgibt. Es werden zwei Implementationen zur Verfügung gestellt, um entweder den Zug durch ein neuronales Netz berechnen oder durch einen menschlichen Spieler auf der Kommandozeile eingeben zu lassen. Während des Spielablaufes fragt das Game abwechselnd die Player nach ihren Zügen und setzt die Steine entsprechend auf das Board. Nach zweimaligem Passen oder bei erreichen des Zuglimits, wird das Spiel beendet und die Punkte gezählt. Zur weiteren Analyse eines Spiels, können Recorder (`recorder_t`) beim Game registriert werden. Diese können den Spielverlauf wahlweise in ASCII-Art oder dem Smart Game Format (SGF)[10] aufzeichnen. Letzteres wird üblicherweise von Go-Software unterstützt, so dass sich Spiele Zug für Zug nachverfolgen lassen.

3.3. Der genetische Algorithmus

Der genetische Algorithmus (Algorithmus 1) imitiert die Evolution um die Spielstärke der neuronalen Netze zu verbessern. Entsprechend wird auch Terminologie aus der Biologie übernommen: Population, Genom Mutation, Generation, Fitness. Eine Generation bezeichnet dabei eine Abfolge von Spielen und die Selektion und Mutation der Population. Die Anzahl dieser Generationen wird anfangs festgelegt.

Jedes Netz stellt hierbei ein Individuum einer Population dar, wobei dessen Fitnesswert davon abhängt, wie viele Spiele dieses Individuum in einer Generation gewonnen hat. Soll die Population eine Generation voran gebracht werden, wird zunächst ausgewählt, welche Individuen in die nächste Generation übernommen werden. Sie werden zufällig ausgesucht, allerdings ist die Wahrscheinlichkeit ein Netz zu wählen proportional zu seinem Fitnesswert. Ist ein Netz also besonders fit, wird es wahrscheinlicher in der nächsten Generation auftauchen. Um keine guten Netze durch "Pech"bei der zufälligen Selektion zu verlieren, werden zusätzlich (in unserem Fall 4) Plätze in der nächsten

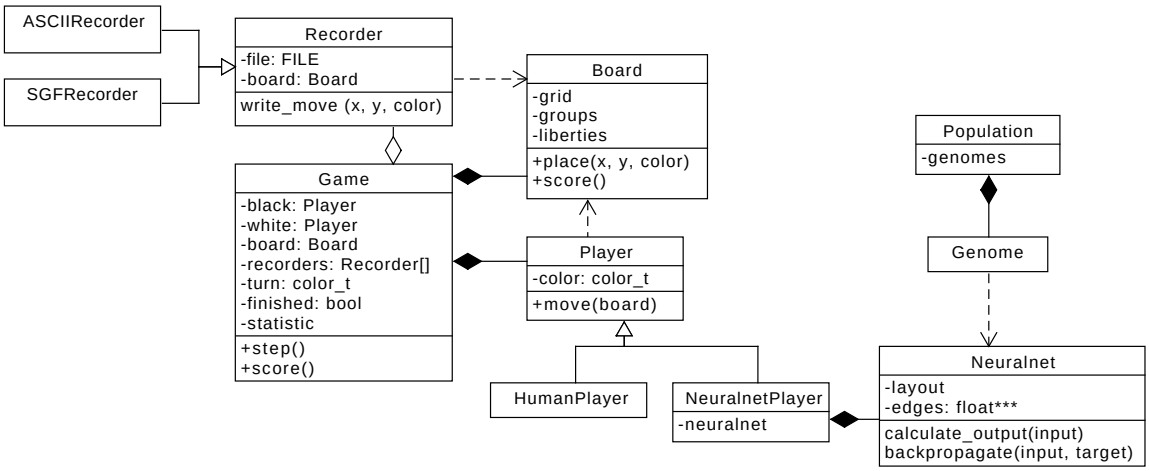

Abbildung 1: Klassendiagramm von Neugengo

Generation reserviert, die mit den unabgeänderten besten Netzen der letzten Generation gefüllt werden. Danach werden die Netze mutiert, ebenfalls zufällig. Es gibt eine bestimmte Mutationswahrscheinlichkeit, mit deren Hilfe überprüft wird, ob ein bestimmtes Kantengewicht mutiert, also um einen zufälligen Wert verändert werden soll, oder nicht. Für jedes Element des Genoms wird eine Zufallszahl zwischen 0 und 1 erzeugt, ist sie kleiner als die Mutationswahrscheinlichkeit, dann wird mutiert. Ist dieser Schritt abgeschlossen, ist die Population eine Generation fortgeschritten. Nun muss die Fitness jedes Netzes durch Spiele gegen andere aktualisiert werden, sodass der Algorithmus wieder von vorne anfangen kann.

Algorithmus 1 Genetischer Algorithmus

Input: Eingabedatei in_file, Iterationen n
Output: Ausgabedatei out_file

1: $N_0 \leftarrow$ LOAD_NETS(in_file)
2: **for** generation $i = 0$ to n **do**
3: $wins \leftarrow (0, 0, \ldots, 0)$
4: **for** $\forall net_a \neq net_b \in N_i$ **do**
5: PLAY_GAME(net_a, net_b)
6: **if** net_a won the game **then**
7: $wins[net_a] \leftarrow wins[net_a] + 1$
8: **else**
9: $wins[net_b] \leftarrow wins[net_b] + 1$
10: **end if**
11: **end for**
12: $N_{i+1} \leftarrow$ THE_NEXT_GENERATION($N_i, wins$)
13: **end for**
14: SAVE_NETS(out_file)

3.4. Die Benutzerschnittstelle

Die Benutzerschnittstelle von Neugengo besteht aus mehreren Command Line Tools, die folgende Funktionen abdecken:

- Erstellung von neuronalen Netzen
- Training von neuronalen Netzen mittels Backpropagation
- Training von neuronalen Netzen mittels genetischen Algorithmus
- Auswertung von Netzen gegeneinander
- Ausführung von Mensch-gegen-Netz-Spielen

Das `ngg_tool` implementiert Funktionen zum zufälligen Erstellen von neuronalen Netzen eines gegebenen Layouts. Weiterhin lassen sich Trainingsdaten generieren, die einer Situation eines Go-Brettes entsprechen. Diese lassen sich verwenden, um die neuronalen Netze mit Backpropagation zu trainieren. Listing 2 (Seite 20) zeigt das Command Line Interface.

`ngg_game` unterzieht eine Menge von neuronalen Netzen einem unsupervised Training mit dem genetischen Algorithmus. Nach einer angegebenen Anzahl an Iterationen oder des Empfangens des Signals `SIGUSR1` werden die neuen bzw. geänderten Netze gespeichert. Siehe Listing 3 (Seite 21).

`ngg_test` kann benutzt werden, um zwei Sets an Netzen gegeneinander spielen zu lassen, wobei jedes Netz zweimal gegen jedes aus dem gegnerischen Set (einmal als Weiß, einmal als Schwarz) spielt. Die Ausgabe enthält die Summe aller Punkte, die beide Sets erzielt haben. Auf diese Weise lassen sich einzelne Netze oder ganze Populationen auf deren Stärke prüfen.

Mit `ngg_hvsai` lassen sich interaktive Spiele gegen abgespeicherte neuronale Netze spielen. Das Spielfeld wird durch ASCII-Art repräsentiert (Abbildung 2). Über eine mit Readline [7] realisierte Eingabe wird eine Position abgefragt, auf diese, falls erlaubt, gesetzt wird. Anschließend wird durch das gewählte neuronale Netz der Zug des Gegners berechnet.

Im Allgemeinen sieht die vorgesehene Nutzung also wie folgt aus: Zuerst werden neuronale Netze und Trainingsdaten erstellt. Dann wird Backpropagation angewandt, um die Netze mit Hilfe der Trainingsdaten zu trainieren. Anschließend können die resultierenden Netze beliebig oft und lange durch den genetischen Algorithmus trainiert werden. Die Ergebnisse werden zur weiteren Verwendung in Dateien abgespeichert. Zur Überprüfung wie erfolgreich das Training verlaufen ist, kann man Netze gegen andere gespeicherte oder zufällig generierte Netze antreten lassen. Weiterhin kann interaktiv gegen die trainierten Spieleauf gespielt werden.

4. Leistungsmessung und -analyse

Die Messung der Rechenzeit erfolgt innerhalb des Programms in der Main Loop. Dadurch wird ausschließlich die Zeit erfasst, in der an dem Ergebnis der Berechnung gearbeitet

```
   0 1 2 3 4 5 6 7 8
0 + + + + + + + + + 0
1 + + + + + + + + + 1
2 + + + + + + + + + 2
3 + + + B + + + + + 3
4 + + + + + + + + + 4
5 + + + + + + + + + 5
6 + W + + + + + + B 6
7 + + + + + B + B W 7
8 + + + + + W + W + 8
   0 1 2 3 4 5 6 7 8

You are black.

Enter x-position (0-8,p): 8
Enter y-position (0-8,p): 8
```

(a) vor einem Zug

```
   0 1 2 3 4 5 6 7 8
0 + + + + + + + + + 0
1 + + + + + + + + + 1
2 + + + + + + + + + 2
3 + + + B + + + + + 3
4 + + + + + + + + + 4
5 + + + + + + + + + 5
6 + W + + + + + + B 6
7 + + + + + B + B + 7
8 + + + + + W + W B 8
   0 1 2 3 4 5 6 7 8
```

(b) nach einem Zug

Abbildung 2: Ausschnitt aus einem interaktivem Spiel mit `ngg_hvsai`

wird. Die Sammlung der aufgezeichneten Statistiken (z. B. die Anzahl der Züge) durch kollektive MPI-Operationen zählt damit nicht zur Rechenzeit.

4.1. Strong Scaling

Für die Bestimmung der Güte der erreichten Parallelisierung wurde das Strong Scaling untersucht, d. h. ein Problem mit gleichbleibender Größe wird mit unterschiedlichen Anzahlen an Prozessen berechnet. Anschließend wurde der Speedup S_p und die Effizienz E_p wie folgt berechnet, wobei p die Anzahl der Prozesse und t_p die mit benötigte Laufzeit ist. t_{seq} entspricht der Laufzeit des sequentiellen Programms.

$$S_p = \frac{t_{seq}}{t_p} \qquad E_p = \frac{S_p}{p}$$

Die Laufzeit ist stark abhängig von zufällig generierten Werten: Zum einen werden die neuronalen Netzwerke zufällig generiert, zum anderen werden diese im Verlauf des Trainings zufällig mutiert. Um vergleichbare Messergebnisse zu erhalten, wurden Vorkehrungen getroffen. In jeder Messung wird das Programm mit den gleichen Netzwerken als Eingabe aufgerufen, zudem wird der Zufallszahlengenerator auf einen bestimmten Wert initialisiert. So kann bei einer gleichbleibenden Anzahl an Iterationen sichergestellt werden, dass bei mehreren Aufrufen jeweils die gleiche Menge an Berechnungen durchgeführt wird. Listing 4 (Seite 21) zeigt ein entsprechendes Batchskript.

4.2. Weak Scaling

Beim Weak Scaling wird die Last mit den proportional zu der Anzahl an Prozessen erhöht, so dass pro Prozess eine fixe Menge an Arbeit zu berechnen ist. Das Weak Scaling zu bestimmen, würde sich schwierig gestalten. Es gibt mehrere Parameter, die die Menge der Berechnung bestimmen: Anzahl der Generationen, Anzahl und Layout der Netze. Durch die verwendeten Zufallszahlen, würde es sehr schwierig sein, anhand dieser Stellschrauben, die Arbeit um einen bestimmten Faktor zu verändern. Die Arbeit ist vor Allem abhängig von der Anzahl berechneter Züge. Würde man mehr Iterationen berechnen, so kann es sein, dass in den Spielen dieser zusätzlicher Generationen im Schnitt weniger oder mehr Züge zu berechnen sind. Auch lässt sich bei geänderter Anzahl oder geändertem Layout (z. B. Größe) der Netze nicht sagen, für wie viele Züge, die hinzugefügten bzw. entfernten oder geänderten Netze verantwortlich sind. Eine Messreihe zum Weak Scaling würde daher nur bedingt verwendbare Ergebnisse liefern.

5. Parallelisierung

Zunächst wurde untersucht, welche Teile der Anwendung für die Parallelisierung in Frage kommen. Profiling ergab, dass ein Großteil der Zeit, wie erwartet, in der Methode `player.move` deren verbracht wird. Dort berechnet das neuronale Netz mit dem momentanen Zustand der Spielfeldes als Eingabe den auszuführenden Spielzug.

Natürlich sind die einzelnen Generationen inherent sequentiell – jede hängt von der vorhergehenden ab – und können daher nicht parallel berechnet werden. Ebenso verhält es sich mit den Zügen innerhalb eines Spieles. Die Spiele innerhalb einer Generation sind voneinander unabhängig, da die neuronalen Netze nur beim Generationenwechsel modifiziert werden. Es ist also möglich, dass das gleiche Netz zeitüberlappend mehrere Spiele spielt.

Im Folgenden bezeichnen p die Anzahl der Prozesse und n die Anzahl der zu berechnenden Spiele.

5.1. Statische Verteilung

Der erste Ansatz war eine gleichmäßige Verteilung, der Spiele auf die Prozesse. Jeder Prozess kennt seinen eindeutigen rank und die Anzahl existierender Prozesse. Daher kann jeder Prozess einfach berechnen, für welche Teilmenge er zuständig ist.

In Abbildung 3 ist zu erkennen, dass die erreichte Effizienz bei 96 Prozessen nur einen Wert von ungefähr 0,4 erreicht. Dies liegt deutlich unter den erwarteten Werten. Eine Analyse mit Vampir [11] (Abbildung 4) zeigt, dass eine Lastungleichheit zwischen den Prozessen herrscht. Diese ist dadurch zu erklären, dass Spiele unterschiedlich lange dauern. Die Länge eines Spiels wurde nach oben beschränkt: Nach 1024 Zügen wird es abgebrochen und die Punkte werden ausgezählt. Mindestens zwei Züge sind notwendig, um ein Spiel zu beenden, da zweifaches Passen zum Spielende führt. Da die neuronalen Netze zufällig erstellt wurden und nach jeder Generation zufällig verändert werden, ist es

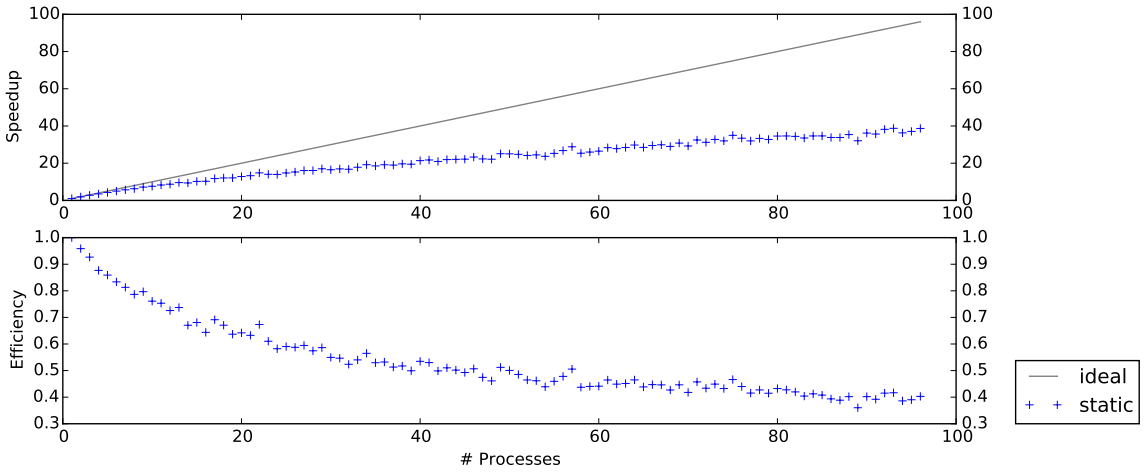

Abbildung 3: Speedup bei statischer Lastverteilung

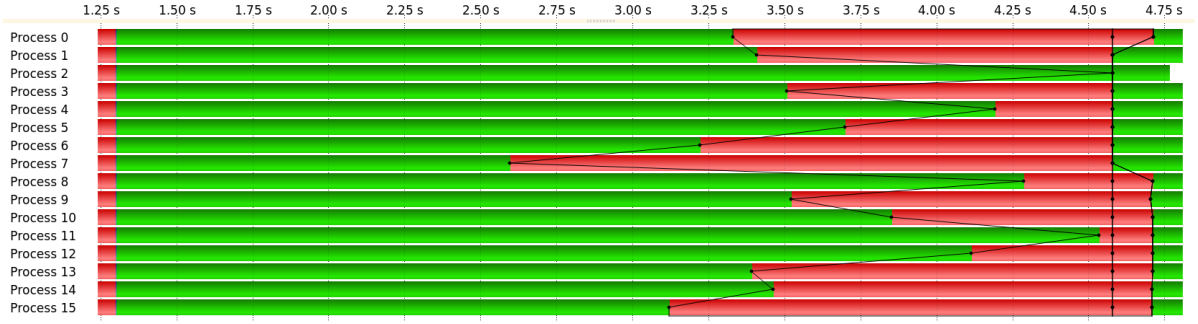

Abbildung 4: Prozess Timeline bei statischer Lastverteilung

uns nicht möglich vorherzusagen, wie lange ein Spiel zwischen zwei bestimmten Netzen dauernd wird.

5.2. Dynamische Verteilung

Gelöst wurde das oben beschriebene Problem der Lastungleichheit durch eine dynamische Verteilung der Spiele auf die Prozesse. Ein Masterprozess übernimmt die Aufgabenverteilung, während die restlichen Workerprozesse, die Berechnungen durchführen. Zunächst wird ein Teil aller Spiele gleichmäßig auf alle Workerprozesse verteilt. Ist ein Worker mit seinem Anteil fertig, schickt er eine Anfrage an den Master. Falls noch Spiele zu berechnen sind, antwortet dieser mit einem Auftrag zu bearbeitender Spiele, andernfalls wird mit einem leeren Auftrag geantwortet. Der Worker bearbeitet den Auftrag oder wartet darauf, dass die anderen ebenfalls fertig werden. Siehe Algorithmen 2 und 3.

Dieser Ansatz wurde mit zwei Parametern untersucht. Zum einen wurde der Anteil der initial verteilten Spiele (*initial*) variiert, zum anderen die Paketgröße (*chunksize*). In der Legende der Diagramme werden diese Konfigurationen in der Form $c-i$ dargestellt, wobei $i \in \mathbb{N} \setminus \{0\}$ die Paketgröße und $i \in [0,1]$ den Anteil statisch verteilter Spiele angibt.

Algorithmus 2 Master

Input: $initial, chunksize, n$ (number of games)

1: $start \leftarrow n \cdot initial$
2: **while** $start < n$ **do**
3: $msg, worker \leftarrow$ RECV($anyone$)
4: SEND($worker, (start, \min(chunksize, n - start))$)
5: $start \leftarrow start + chunksize$
6: **end while**
7: **for** each process p **do**
8: $msg, worker \leftarrow$ RECV($anyone$)
9: SEND($worker, (0, 0,$ "finished"))
10: **end for**

5.2.1. Paketgröße

Da die Anzahl an Anfragen mit sinkender Paketgröße steigt, wurde vermutet, dass es einen Punkt gibt, ab dem die weitere Verkleinerung der Pakete für mehr Overhead sorgt, als dass sie Zeit spart. In Abbildung 5 sind die Speedupkurven für Paketgrößen von 1, 5 und 10 Spielen aufgezeichnet, während 50 % der Spiele gleichmäßig verteilt wurden. Es ist zu erkennen, dass bei der feinstmöglichen Granularität von einem Spiel pro Paket, die erreichte Effizienz am größten ist.

Algorithmus 3 Worker

Input: $initial, chunksize, n$ (number of games)

1: $start, chunksize \leftarrow$ PARTITION$(n \cdot initial)$
2: **while** $chunksize \neq 0$ **do**
3: **for** $g \in [start, start + chunksize)$ **do**
4: calculate game $\#g$
5: count the $(wins)$
6: **end for**
7: SEND$(master$, "request work"$)$
8: $start, chunksize \leftarrow$RECV$(master)$
9: **end while**

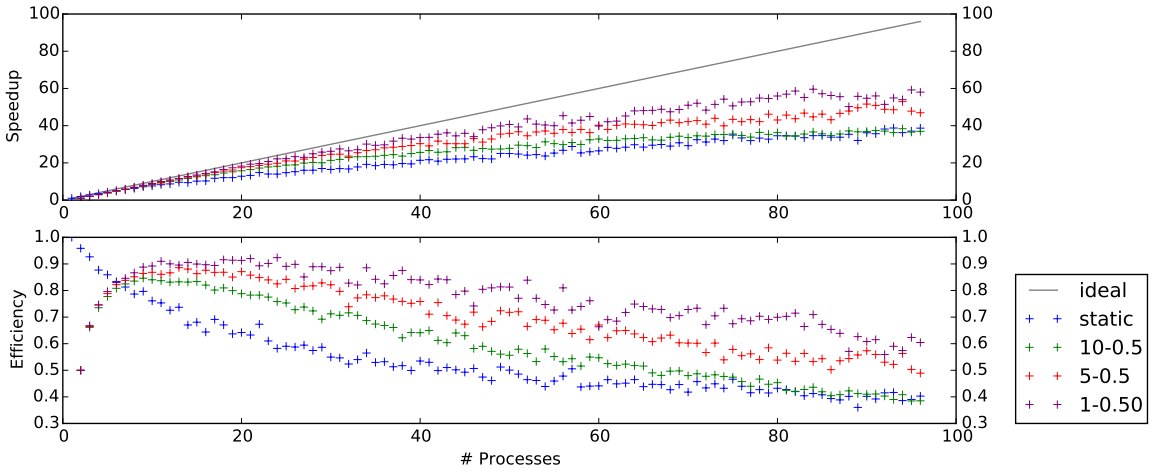

Abbildung 5: Vergleich von Paketgrößen

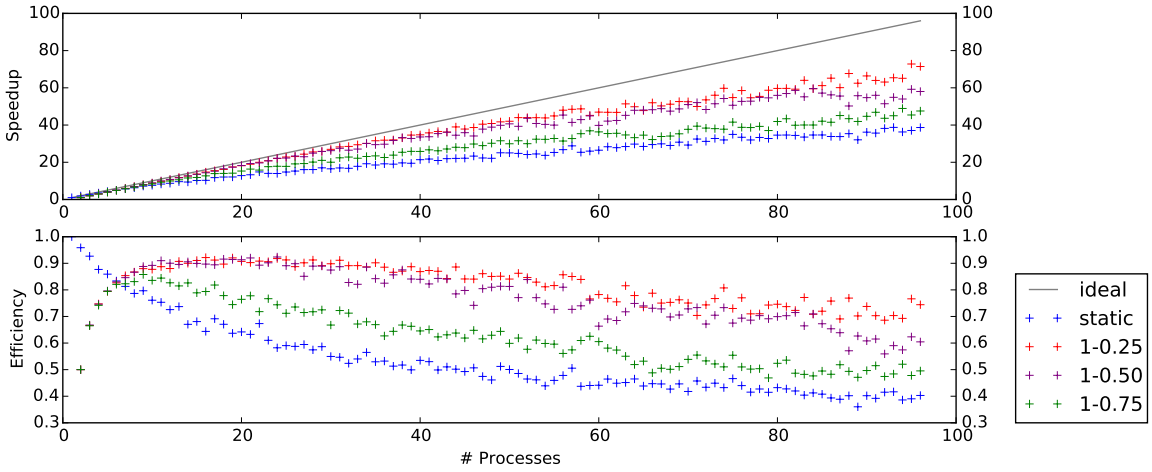

Abbildung 6: Vergleich von Anteilen der initial verteilten Spiele

5.2.2. Anteil statischer Verteilung

Der zweite Parameter gibt an, wie viele Spiele zu Beginn einer Generation gleichmäßig auf die Worker aufgeteilt werden. Je größer der Anteil ist, desto weniger muss über MPI kommuniziert werden. Abbildung 6 zeigt den Speedup für die initiale Verteilung von 25 % bis 75 %. Je weniger statisch verteilt wurde, desto effizienter ist die Parallelisierung. Aus Gründen der Übersichtlichkeit wurden nur drei Konfigurationen im Diagramm aufgezeichnet. Durch Messungen wurde jedoch festgestellt, dass initialen Verteilungen von maximal einem Drittel der Spiele kein wesentlicher Unterschied mehr erreicht wurde.

5.2.3. Vergleich von dynamischer und statischer Verteilung

Während der erste Ansatz beinahe ohne MPI-Kommunikation auskommt – es werden ausschließlich die Ergebnisse am Generationsende aufsummiert – benötigt die dynamische Verteilung pro verteiltem Paket zwei Punkt-zu-Punkt Übertragungen. Zudem beteiligt sich der Masterprozess nicht an der Berechnung. Dennoch kommt es zu einem deutlichen Unterschied bezüglich Speedup und Effizienz (Abbildung 7): Mit ca. 75 % Effizienz bei 96 Prozessen erreicht die dynamische Verteilung ein wesentlich besseres Ergebnis als die statische Aufteilung mit ca. 40 %.

In Abbildung 8 ist das Kommunikationsschema einer Generation dargestellt. Der Masterprozess (Prozess 0) beantwortet ab 1,8 s die eintreffenden Requests der Workerprozesse. Jeder erkennbare Strich zwischen dem Master und einem Worker repräsentiert eine Request und die dazugehörige Response.

Gut erkennbar sind die unterschiedlichen Längen der Spiele. Die grün dargestellten Bereiche zwischen den MPI-Aufrufen auf den Balken der Worker zeigen jeweils die Berechnung eines Spiels. Betrachtet man beispielsweise den Prozess 15. Das erste vom Master zugeteilte Spiel wird ab 2,1 s berechnet und braucht ungefähr 0,2 s. Anschließend bei ca. 2,3 s, kommuniziert Prozess 15 zweimal kurz hintereinander mit dem Master. In

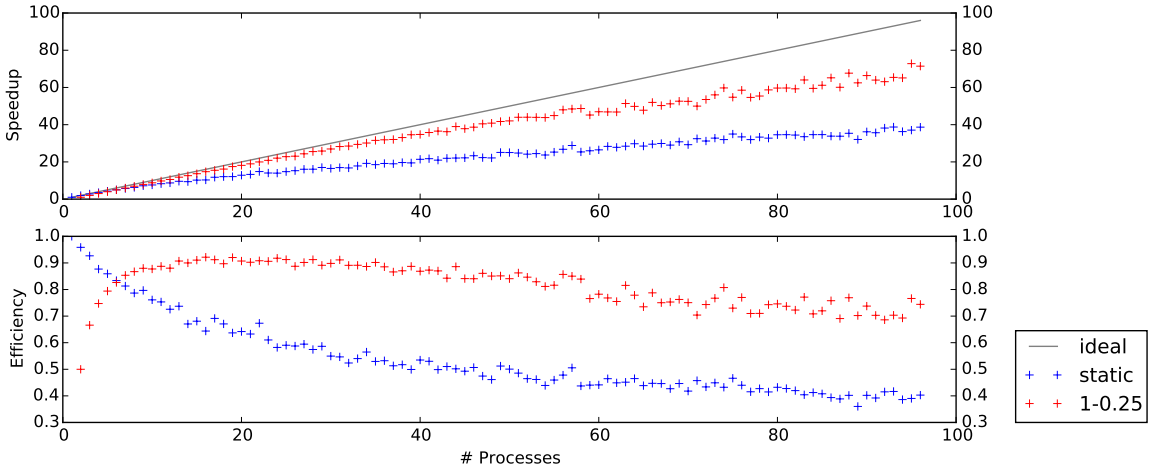

Abbildung 7: Vergleich von dynamischer und statischer Lastverteilung

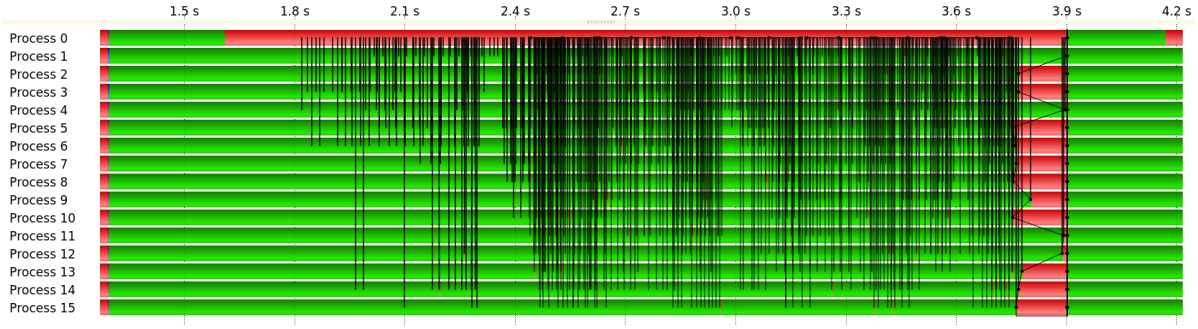

Abbildung 8: Prozess Timeline bei dynamischer Lastverteilung

dem Intervall zwischen den beiden Anfragen wird ein weiteres Spiel berechnet.

Die Antwortzeit bei den Anfragen ist im Vergleich zu der Zeit, welche für die Berechnung eines Spiels aufgewendet wird, so gering, dass die Wartezeit für die Worker vernachlässigbar gering bleibt. Auch die Wahrscheinlichkeit für gleichzeitige Anfragen von mehreren Prozessen ist gering genug, so dass mit synchronen Sende- und Empfangsinstruktionen gearbeitet werden kann.

Die Lastungleichheit ist weitestgehend behoben. Zum Ende der Generation muss zwar noch gewartet werden, jedoch ist die Ausnutzung der verfügbaren Rechenzeit im Vergleich zu der vorherigen Situation deutlich besser.

5.3. Hybride Parallelisierung

Zusätzlich zu obigem Parallelisierungsschema wurde hybride Parallelisierung mit MPI und OpenMP ausprobiert. Dabei wurde untersucht, wie sich die Aufteilung der Schleifen in der Ausgabeberechnung der neuronalen Netz auf verschiedene Threads verhält.

In der Testkonfiguration wurde ein rechnender MPI-Prozess mit OpenMP-Threads gestartet, so dass jeder Thread auf einem eigenen Prozessorkern läuft. Mit einem einzelnen Thread – also sequentiell – lag die Laufzeit deutlich über der, der entsprechenden Variant ohne OpenMP. Mit jedem zusätzlichen Thread stieg die Laufzeit weiter an. Daher wurde dieser Ansatz wieder verworfen und sich auf einen MPI-Prozess pro Prozessorkern beschränkt.

Zwar wird viel Rechenzeit in `nnnet_calculate_output` verbracht, aber wir vermuten, dass Berechnung einer einzelnen Ausgabe schnell genug ist, so dass der Overhead beim Erstellen und Zerstören der Threads gegenüber der eingesparten Rechenzeit deutlich überwiegt. Da bei jedem Zug eine Ausgabe berechnet wird resultieren viele und lange Spiele einer anteilig hohen Benutzung dieses Codeabschnitts.

6. Die Ergebnisse des Trainings

Das bisherige Ergebnis war etwas ernüchternd. Die trainierten Netze spielen in der Regel besser, als untrainierte. Von zehn trainierten Netzen hatte das beste eine Gewinnquote von ca. 87 % gegen nicht trainierte Netze, das schlechteste eine von ca. 45 % und der Durchschnitt aller Netze lag bei ca. 67 % (Tabelle 1, Seite 22). Doch selbst das beste Netz (egal ob gemessen an der Gewinnquote gegen untrainierte oder gleich trainierte) hat keine Chance gegen einen Menschen, sofern dieser halbwegs überlegt spielt.

Für die Tragweite von neuronalen Netzen, im Hinblick auf die Kompetenz Go gut spielen zu können, hat dies allerdings wenig zu Bedeuten. In diesem Projekt haben wir lediglich eine von vielen Arten von neuronalen Netzen benutzt, wobei nicht klar ist, ob diese Art die best geeignetste ist. Zusätzlich dazu gibt es in diesem Ansatz viele Parameter, die durch teilweise Abänderung mit Sicherheit einen positiven Effekt auf die Performance der Netze hätten.

7. Fazit

Die Spielleistung der trainierten neuronalen Netze war nicht überragend. Für ein besseres Ergebnis in dieser Hinsicht, hätte man sich mehr mit der Theorie hinter neuronalen Netzen beschäftigen müssen. Auch weitere Erfahrung in Bezug auf Computer Go wäre sicherlich nützlich gewesen.

Da der Titel des Praktikum „Parallele Programmierung“ lautet, war die Beschäftigung mit dem Thema der künstlichen Intelligenz nur ein Nebenprodukt. Die im Vordergrund stehende Parallelisierung des Programms war insofern ein Erfolg, dass wir ein naives, nur mäßig effizientes, Parallelisierungsschema analysiert und die Schwachstellen erkannt haben. Mittels den gewonnenen Erkenntnissen wurde erfolgreich ein alternatives Schema entworfen und implementiert. Dieses lieferte einen, in unseren Augen, akzeptablen Speedup. Interessant an der Parallelisierung war, dass sich die zu verteilende Last nicht vorhersehbar änderte und daher dynamisch, während der Laufzeit darauf eingegangen werden musste.

Anhänge

A. Verwendete Software

Für unsere Anwendung haben wir die folgende Software verwendet:

Compiler

- GNU Compiler Collection [6]

Libraries

- GNU C Library [5]
- OpenMPI [9]
- GNU Readline [7]
- Check [1]

Tools

- Git [4]
- CMake [2]
- Doxygen [3]
- Vampir [11]

Dieser Bericht wurde mit LaTeX und KOMA-Script erstellt. Die Diagramme wurden mit matplotlib [8] erstellt.

B. Dateiformat

```
1  # nnet_set_t
2  size = 2
3  # ----------------------------------
4  # neuralnet #0
5  # neuralnet_t
6  edge_count = 17
7  layer_count = 3
8  # neurons_per_layer
9  2 3 2
10 # edges from layer 0 to 1
11 -6.766385436e-01 +7.558403015e-01 +2.843139172e-01
12 +5.824345350e-01 +4.133069515e-02 +6.128995419e-01
```

```
-2.546305060e-01 -5.267066956e-01 -6.954469681e-01
# edges from layer 1 to 2
+4.708336592e-01 -3.204008937e-01
+2.103395462e-01 -5.273654461e-01
-6.096124649e-02 +8.068499565e-01
-9.861236215e-01 +6.316629648e-01
# ------------------------------
# neuralnet #1
# neuralnet_t
edge_count = 17
layer_count = 3
# neurons_per_layer
2 3 2
# edges from layer 0 to 1
-1.220832467e-01 +9.980778694e-01 -4.880219698e-02
+1.977531910e-01 -2.914841175e-01 +8.421014547e-01
+8.383474350e-01 -7.447167635e-01 +9.050107002e-02
# edges from layer 1 to 2
-1.040073037e-01 +3.890092373e-01
+1.624392271e-01 -4.180262089e-01
-1.358566880e-01 +4.858006239e-01
-6.621859670e-01 -8.515428305e-01
```

Listing 1: Ausgabedatei von `ngg_tool -a create -layout "2 3 2" -o filename`

C. Command Line Interfaces

```
\$ ./ngg_tool --help
Usage: ngg_tool [OPTION...]

  -a, --action=STRING        create: creates a new neural network
                             gen-data: generates training data
                             train: trains a neural network with
                             supervised learning
                             calc: calculates output of a neural network
                             given input
  -b, --binary               use binary files
      --binary-in            use binary input file
      --binary-out           use binary output file
  -i, --in=FILE              load neuralnet(s) from file
  -l, --layer=NUMS           number of neurons in each layer
                             e.g. "2 3 3 2"
  -n, --number=NUM           number of networks to create or training
```

```
                              iterations
  -o, --out=FILE              output neuralnet(s) to file
  -s, --board-size=NUM        size of the used go board
  -t, --training-data=FILE    training data to use
  -v, --netver=0              nnet player version
  -?, --help                  Give this help list
      --usage                 Give a short usage message
```

Listing 2: Interface von ngg_tool

```
\$ ./ngg_game --help
Usage: ngg_game [OPTION...]

  -b, --binary                use binary files
      --binary-in             use binary input file
      --binary-out            use binary output file
  -h, --human-readable        human readable output, no csv
  -i, --in=FILE               load neuralnet from file
  -n, --number=NUM            number of generations
  -o, --out=FILE              output neuralnet to file
  -s, --board-size=NUM        size of the used go board
      --sched-chunksize=int
      --sched-initial=double
      --seed=decimal          seed for the random number generator
  -v, --verbose               more prints more information
  -?, --help                  Give this help list
      --usage                 Give a short usage message
```

Listing 3: Interface von ngg_game

D. Skripte

```
#!/bin/sh
#SBATCH -N8
#SBATCH --ntasks-per-node 12

# script for measuring strong scaling
# call from build/src/
# first cli parameter has to be a file containing
# a set of neural networks
# output:
# num procs, last generation, time, games, plays, passes

```

```
input_nets="${1}"

for i in $(seq 2 96); do
  line="$i,$(
    mpirun -np "$i" ./ngg_game -i "$input_nets" \
                               -o /dev/null \
                               -s 9 \
                               -n 10 \
                               --seed 42 \
                               --sched-initial 0.25 \
                               --sched-chunksize 1 \
    | grep "total" \
    | cut -d " " -f3
  )"
  echo "$line"
done
```

Listing 4: Batchskript zur Messung des Strong Scaling

E. Spielergebnisse

Population	Siege	Niederlagen	Unentschieden
1	19	12	1
2	17	12	3
3	26	4	2
4	19	11	2
5	24	8	0
6	26	6	0
7	18	11	3
8	18	9	5
9	22	10	0
10	13	16	3

Tabelle 1: Spielergebnisse des jeweils besten Netzes nach 10 000 Generationen unsupervised Training aus 10 Populationen gegen 16 untrainierte Netze

Literatur

[1] *Check*. URL: `http://check.sourceforge.net/` (besucht am 29. 09. 2015).

[2] *CMake*. URL: `https://cmake.org/` (besucht am 29. 09. 2015).

[3] *Doxygen*. URL: `http://www.stack.nl/~dimitri/doxygen/` (besucht am 29. 09. 2015).

[4] *Git*. URL: `https://git-scm.com/` (besucht am 29. 09. 2015).

[5] *GNU C Library*. URL: `https://www.gnu.org/software/libc/` (besucht am 29. 09. 2015).

[6] *GNU Compiler Collection*. URL: `https://gcc.gnu.org/` (besucht am 29. 09. 2015).

[7] *GNU Readline*. URL: `https://cnswww.cns.cwru.edu/php/chet/readline/rltop.html` (besucht am 29. 09. 2015).

[8] *matplotlib*. URL: `http://matplotlib.org/` (besucht am 29. 09. 2015).

[9] *OpenMPI*. URL: `https://www.open-mpi.org/` (besucht am 29. 09. 2015).

[10] *SGF File Format FF[4]*. 6. Aug. 2006. URL: `http://www.red-bean.com/sgf/` (besucht am 30. 09. 2015).

[11] *Vampir*. URL: `https://www.vampir.eu/` (besucht am 29. 09. 2015).

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmen

Listings