

NeuGenGo

Kann unser neuronales Netz besser Go spielen als wir?

Lennart Braun, Armin Schaare, Theresa Eimer

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik, Arbeitsbereich WR
Praktikum Parallele Programmierung SS 15

9. September 2015

- 1 Problemstellung
- 2 Lösungsansatz
- 3 Parallelisierungsschema
- 4 Ergebnis

Problemstellung

Ziel:

- Neuronale Netzwerke zu trainieren, so dass diese uns im Go schlagen können.

Zwischenziel:

- Netze zu trainieren, so dass sie besser als zufällig erzeugte Netze spielen.

Go

- Asiatisches Brettspiel
- Wird auf Brettern mit 19×19 Knoten gespielt.
- Ziel: Gebiet einkreisen und gegnerische Steine schlagen
- Spielende: wenn beide Spieler passen

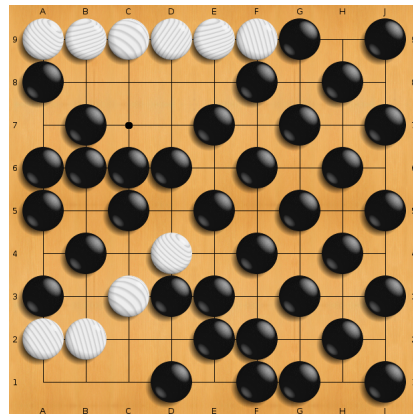


Abbildung: generated with qGo

Neuronale Netzwerke

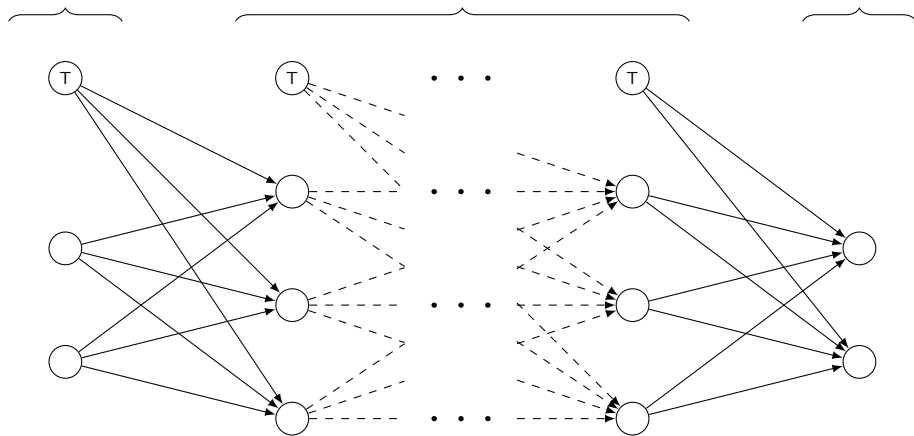
- Besteht aus mehreren Schichten (Layer)
- Layer bestehen aus Neuronen
- Neuronen benachbarter Layer sind alle durch Kanten untereinander verbunden
- Neuronen berechnen ihre Werte durch Aufsummieren aller eingehenden Kantengewichte multipliziert mit den Werten an den ausgehenden Neuronen
- Sigmoid Funktion wird auf das Ergebnis angewandt, sodass alle Werte zwischen 0 und 1 sind.

Neuronale Netzwerke

Input Layer

Hidden Layer

Output Layer



Lösungsansatz

- Beschränkung auf 9×9 Bretter
- Feedforward Netze:
5 Layer, mit folgender Anzahl an Neuronen: 81 82 82 82 82
- Genetische Algorithmen:
Mutationsrate: 0.5%
Netze pro Population: 32

Lösungsansatz

Algorithmus 1 sequentielle Lösung

```
1:  $N_0 \leftarrow \{n \text{ zufällig generierte neuronale Netzwerke} \}$ 
2: for  $net \in N_0$  do
3:   trainiere  $net$  auf regelgerechtes Spielen
4: end for
5: for Generation  $i = 0$  bis  $\dots$  do
6:   for  $\forall net_a \neq net_b \in N_i$  do
7:     lass  $net_a, net_b$  gegeneinander spielen
8:     zähle die Anzahl der Siege
9:   end for
10:  generiere  $N_{i+1}$  mittels genetischen Algorithmus abhängig von  $N_i$  und den Spielergebnissen
11: end for
12: Speichere  $N_n$ 
```

UML

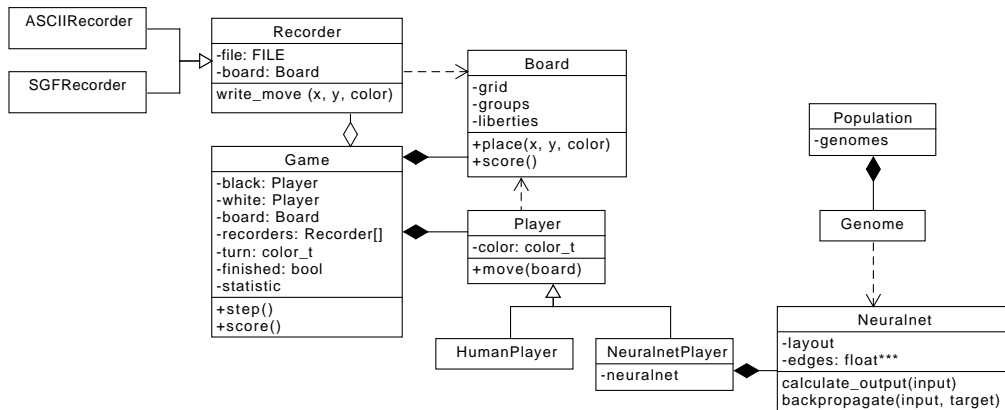


Abbildung: Klassendiagramm

Parallelisierungsschema

Was ist parallelisierbar?

- Die Generationen sind inherent sequentiell
- + die einzelnen Spiele sind unabhängig voneinander (**for** $net_a, net_b \in N_i$ **do**)
- ? die Ausgabeberechnung in den Neuronalen Netzwerken (dreifache Schleife)

Parallelisierung der Spielphase

Algorithmus 2 parallele Spielphase (1)

```

1: for Generation  $i = 0$  bis ... do
2:   for  $\forall net_a \in N_i$  pardo
3:     for  $\forall net_b \neq net_a \in N_i$  pardo
4:       lass  $net_a, net_b$  spielen
5:       zähle die Siege ( $wins$ )
6:     end pardo
7:   end pardo
8:   reduce( $wins$ )
9:   if rank = 0 then generiere  $N_{i+1}$  end if
10:  broadcast( $N_i, 0$ )
11: end for

```

Ziel: n^2 Spiele auf p Prozesse zu verteilen

- Master erstellt N_{i+1} .
- Master sendet N_{i+1} an alle.
- Gleichmäßige Verteilung der inneren Schleifen (Zeilen 3,4).

Parallelisierung der Spielphase

Algorithmus 3 parallele Spielphase (1)

```

1: for Generation  $i = 0$  bis ... do
2:   for  $\forall net_a \in N_i$  pardo
3:     for  $\forall net_b \neq net_a \in N_i$  pardo
4:       lass  $net_a, net_b$  spielen
5:       zähle die Siege ( $wins$ )
6:     end pardo
7:   end pardo
8:   reduce( $wins$ )
9:   if rank = 0 then generiere  $N_{i+1}$  end if
10:  broadcast( $N_i, 0$ )
11: end for

```

Ziel: n^2 Spiele auf p Prozesse zu verteilen

- Master erstellt N_{i+1} .
- Master sendet N_{i+1} an alle.
- Gleichmäßige Verteilung der inneren Schleifen (Zeilen 3,4).

Probleme:

- ≈ 100 KiB pro Netzwerk
- viele kollektive Operationen

Parallelisierung der Spielphase

Algorithmus 4 parallele Spielphase (2)

```

1: for Generation  $i = 0$  bis ... do
2:   for  $\forall net_a \in N_i$  pardo
3:     for  $\forall net_b \neq net_a \in N_i$  pardo
4:       lass  $net_a, net_b$  spielen
5:       zähle die Siege ( $wins$ )
6:     end pardo
7:   end pardo
8:   reduce( $wins$ )
9:   generiere  $N_{i+1}$ 
10: end for

```

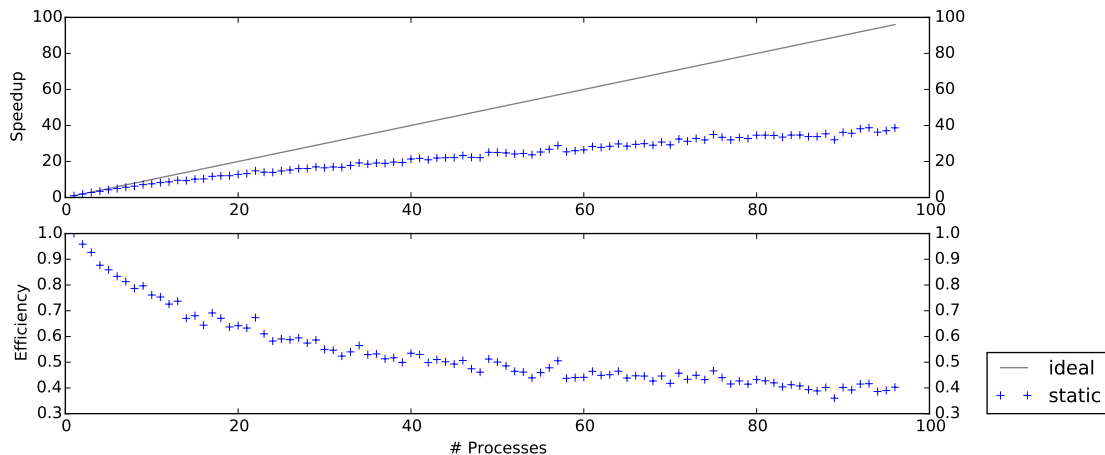
Ziel: n^2 Spiele auf p Prozesse zu verteilen

- Jeder erstellt N_{i+1} .
- ~~Master sendet N_{i+1} an alle.~~
- Gleichmäßige Verteilung der inneren Schleifen (Zeilen 3,4).

Probleme:

- ~~≈ 100 KiB pro Netzwerk~~
- ~~viele kollektive Operationen~~
- ?

Not So Strong Scaling



Not So Strong Scaling

Spurdatenanalyse



Abbildung: Vampir

Not So Strong Scaling

Was ist da los?

- Spiele dauern unterschiedlich lange (2-1024 Züge)
- Länge ist nicht vorhersagbar

⇒ Lastungleichheit zwischen den Prozessen

Not So Strong Scaling

Was ist da los?

- Spiele dauern unterschiedlich lange (2-1024 Züge)
- Länge ist nicht vorhersagbar

⇒ Lastungleichheit zwischen den Prozessen

Lösung: Dynamisches Scheduling

- Master/Worker Modell
- ein Anteil der Spiele wird gleichmäßig verteilt (*initial*)
- Master verteilt restliche Spiele paketweise an idlende Prozesse (*chunksize*)

Dynamic Scheduling

Algorithmus 5 Master

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

```
1: start  $\leftarrow n \cdot \textit{initial}$ 
2: while start < n do
3:   msg, p  $\leftarrow \text{Recv}()$ 
4:   Send(p, (start, chunksize))
5:   start  $\leftarrow \textit{start} + \textit{chunksize}$ 
6: end while
7: for each process p do
8:   msg, p  $\leftarrow \text{Recv}()$ 
9:   Send(p, (0, 0, "nothing to do"))
10: end for
```

Dynamic Scheduling

Algorithmus 7 Master

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

```

1: start  $\leftarrow n \cdot \textit{initial}$ 
2: while start < n do
3:   msg, p  $\leftarrow$  Recv(?)
4:   Send(p, (start, chunksize))
5:   start  $\leftarrow$  start + chunksize
6: end while
7: for each process p do
8:   msg, p  $\leftarrow$  Recv(?)
9:   Send(p, (0, 0, "nothing to do"))
10: end for

```

Algorithmus 8 Worker

Input: *initial*, *chunksize*, number of games

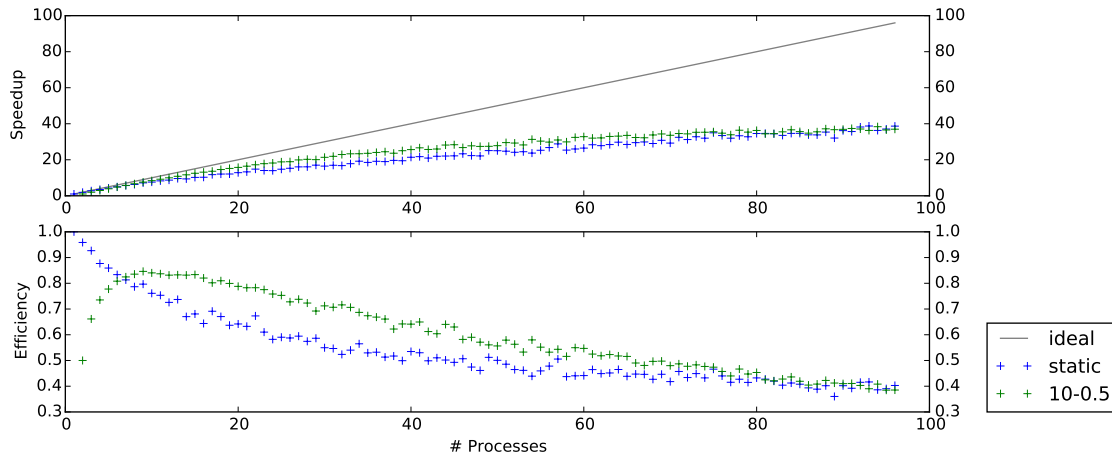
```

1: start, chunksize  $\leftarrow$  partition(n · initial)
2: while chunksize  $\neq$  0 do
3:   for g  $\in$  [start, start + chunksize) do
4:     rechne Game #g
5:     zähle die Siege (wins)
6:   end for
7:   Send(master, "I'm bored")
8:   start, chunksize  $\leftarrow$  Recv(master)
9: end while

```

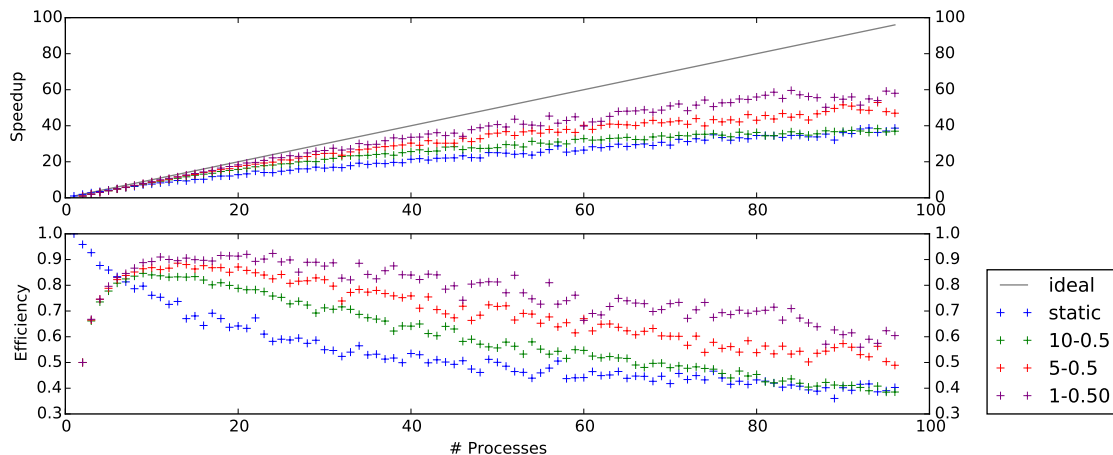
Stronger Scaling

Dynamic Scheduling



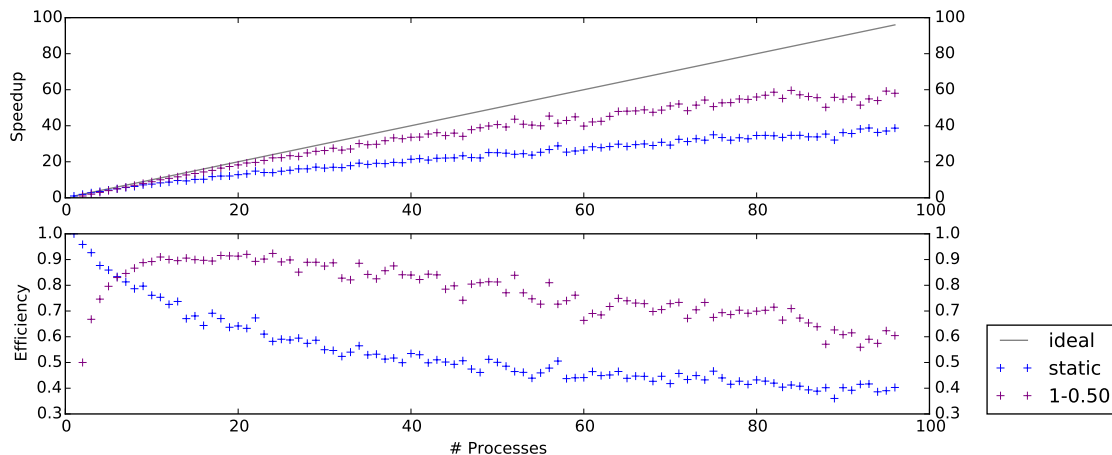
Stronger Scaling

Chunksize



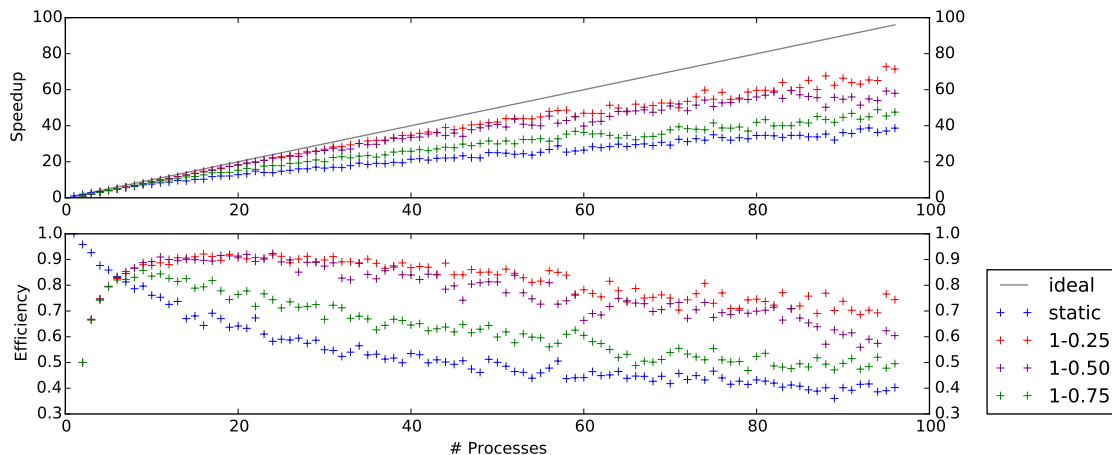
Stronger Scaling

Initial

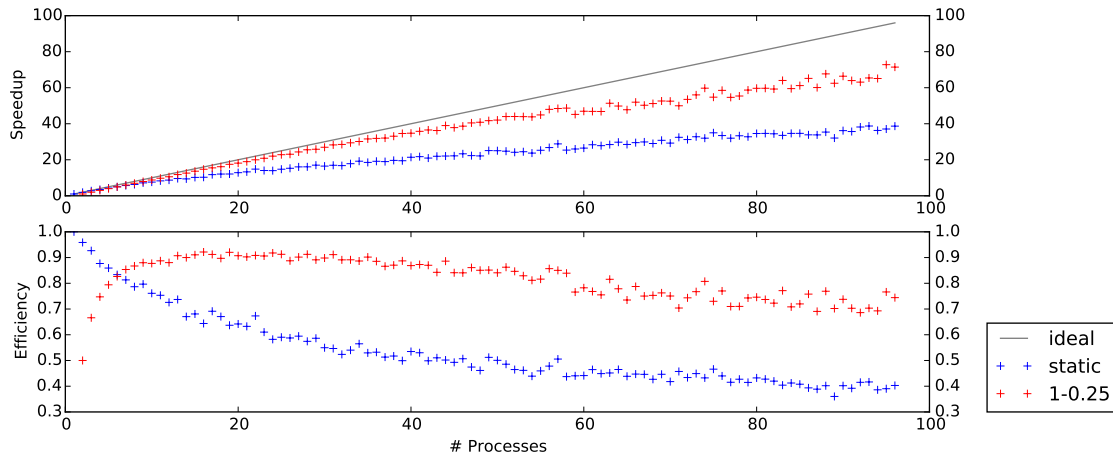


Stronger Scaling

Initial



Much Stronger Scaling



Much Stronger Scaling

Spurdatenanalyse

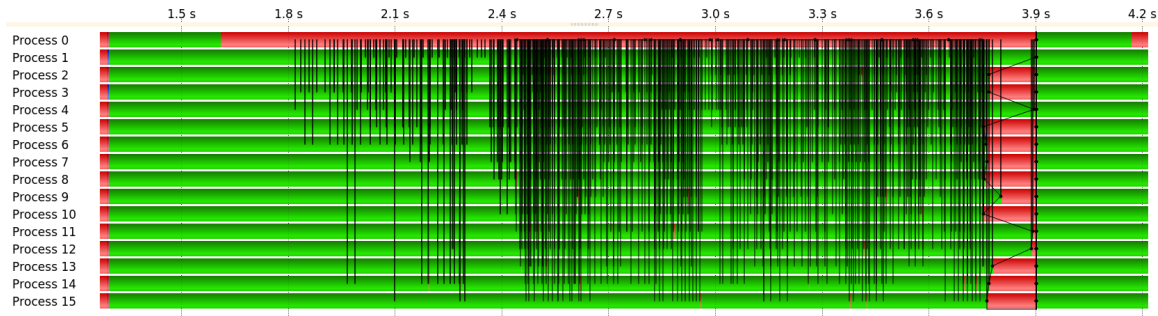


Abbildung: Vampir

Parallelisierung der neuronalen Netzwerke (OpenMP)

Algorithmus 9 output calculation

Input: in

Output: out

```
1: for each gap do
2:   init(out)
3:   for to  $\leftarrow$  0 to neurons_per_layer[gap+1] pardo
4:     for from  $\leftarrow$  0 to neurons_per_layer[gap] do
5:       out[to]  $\leftarrow$  out[to] +
6:         in[from] * edges[gap][from][to]
7:     end for
8:   end pardo
9:   swap(in, out)
10: end for
```

Parallelisierung der neuronalen Netzwerke (OpenMP)

Algorithmus 10 output calculation

Input: in

Output: out

```
1: for each gap do
2:   init(out)
3:   for to  $\leftarrow 0$  to neurons_per_layer[gap+1] pardo
4:     for from  $\leftarrow 0$  to neurons_per_layer[gap] do
5:       out[to]  $\leftarrow$  out[to] +
6:         in[from] * edges[gap][from][to]
7:     end for
8:   end pardo
9:   swap(in, out)
10: end for
```

- Ist langsam.
- Je mehr Threads desto langsamer.
- Vermutlich zu hoher Overhead durch Fork/Join bei wenig Iterationen.

Funktioniert das Training?

- Antwort: Teilweise.
- Gegen zufällige Netze gewinnen trainierte Netze in ca. 60% der Fälle
- Gegen menschliche Spieler haben trainierte Netze keine Chance, sofern der Spieler überlegt Steine setzt.

Warum ist das Training nur mäßig erfolgreich?

Vermutungen:

- Zu viele Kanten
- Netze werden nie gegen menschliche Spieler evaluiert und verbessert.
- Schlechte Parameterwahl bei Feedforward-Netzen und/oder beim genetischen Algorithmus.

Zahlen und so

- <https://github.com/lenerd/papo-project>
- GNU GPLv3?
- 512 Commits
- 6500 Zeilen C Code
- 500 000 Generationen
- Spaß.