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

Problemstellung

- Unser Ziel ist es, neuronale Netzwerke zu trainieren, so dass diese uns im Go schlagen können.
- Zwischenziel / Alternative: Können wir neuronale Netze so trainieren, so dass sie besser als zufällig erzeugte Netze spielen?

TODO: Ziele besser verkaufen

Go

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

TODO: Graphik (möglichst unter CC / selbst erstellt)

Abbildung: Stellung eines Go Spiels

Neuronale Netzwerke

- TODO: kurze Beschreibung von neuronalen Netzwerken

TODO: Graphik (möglichst unter CC / selbst erstellt)

Abbildung: Schema eines neuronalen Netzwerks

Lösungsansatz

- Beschränkung auf 9×9 Bretter
- Feedforward Netze (TODO: Layout)
- Genetische Algorithmen (TODO: Parameter)

Lösungsansatz

Algorithmus 1 sequentielle Lösung

1: $N_0 \leftarrow \{n$ 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 ... **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

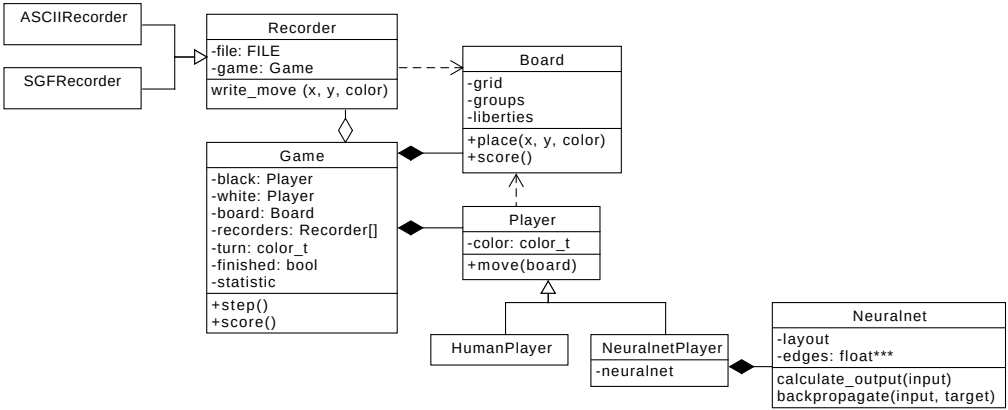


Abbildung: Klassendiagramm

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 ∀net_a ∈ N_i pardo
 3:         for ∀net_b ≠ net_a ∈ 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 3 parallele Spielphase (2)

```
 1: for Generation i = 0 bis ... do
 2:     for ∀net_a ∈ N_i pardo
 3:         for ∀net_b ≠ net_a ∈ 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
```

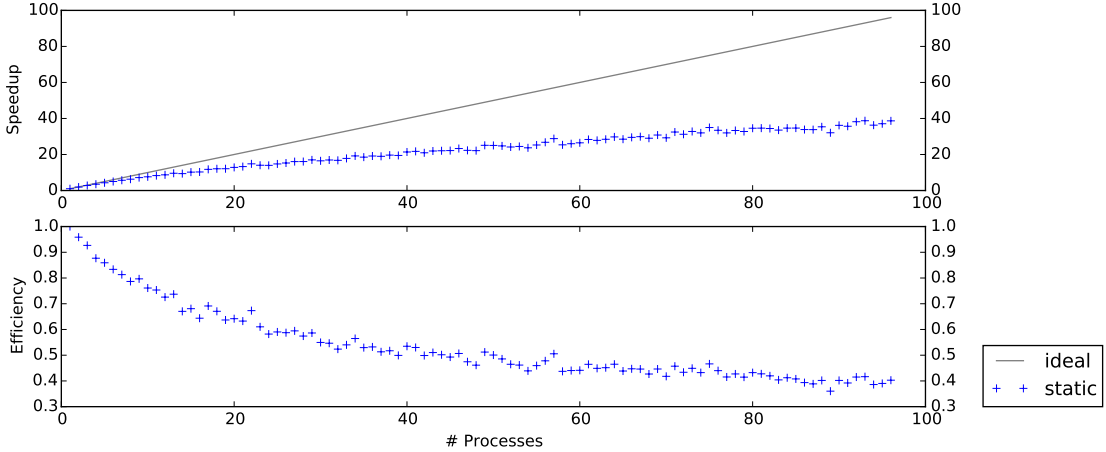
Ziel: n^2 Spiele auf p Prozesse zu verteilen

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

Probleme:

- ~~$\approx$ 27 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

$\Rightarrow$ 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 4 Master

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

```
1:  start ← n · initial
2:  while start < n do
3:      msg, p ←Recv(?)
4:      Send(p, (start, chunksize))
5:      start ← start + chunksize
6:  end while
7:  for each process p do
8:      msg, p ←Recv(?)
9:      Send(p, (0, 0, "nothing to do"))
10: end for
```

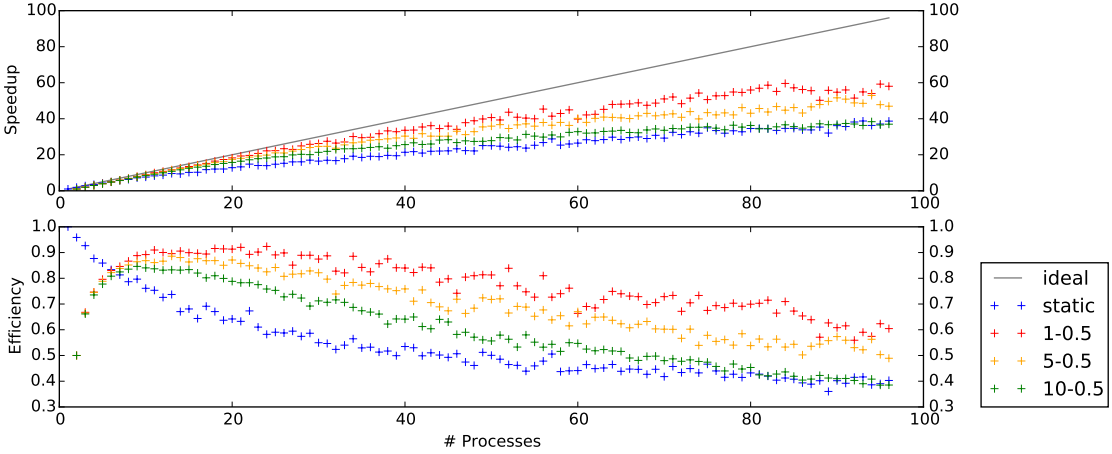
Algorithmus 5 Worker

Input: initial, chunksize, number of games

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

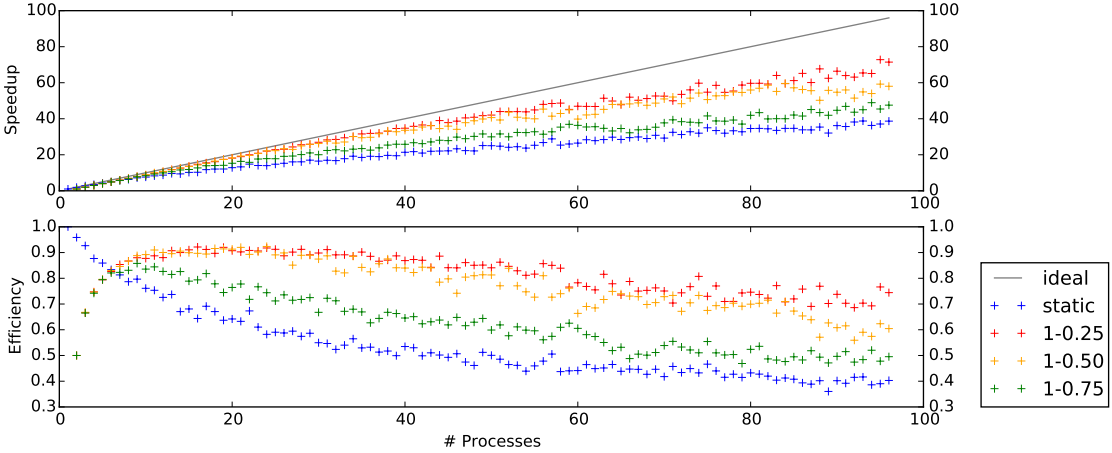
Stronger Scaling

Chunksize

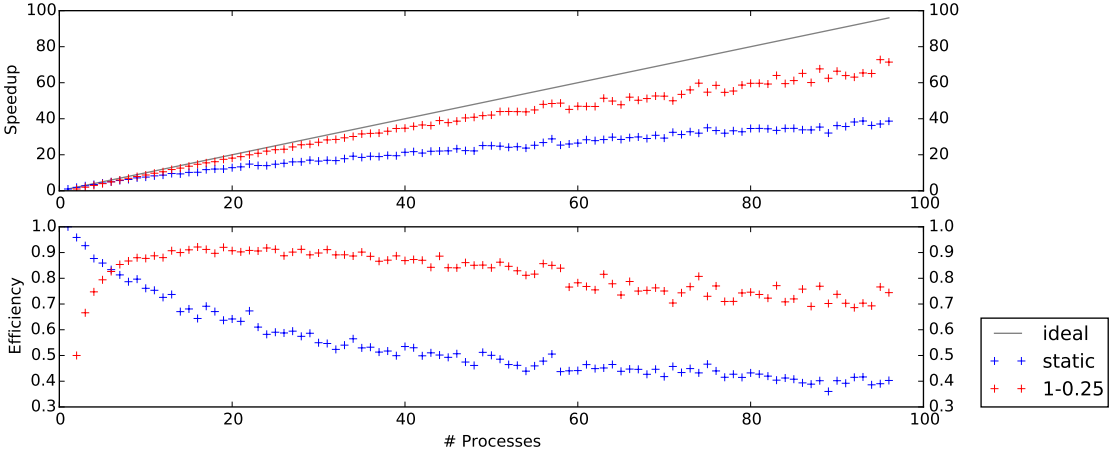


Stronger Scaling

Initial



Much Stronger Scaling



Much Stronger Scaling

Spurdatenanalyse

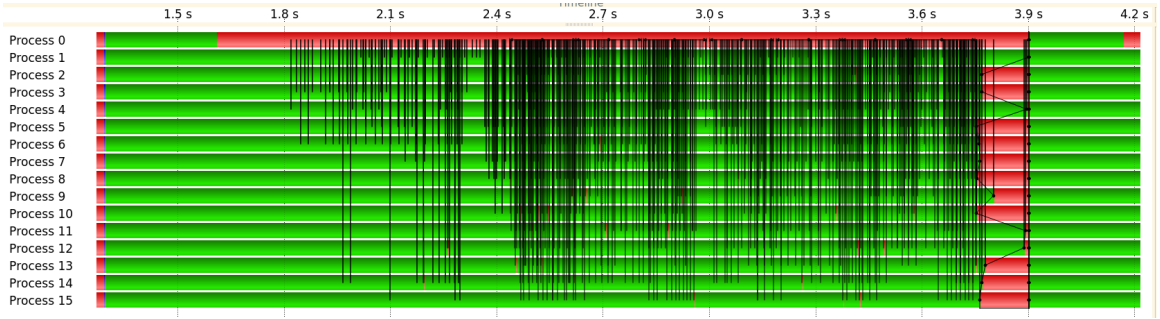


Abbildung: Vampir

Parallelisierung der neuronalen Netzwerke (OpenMP)

Algorithmus 6 output calculation

Input: in
Output: out

```
1:  for each gap do
2:      init(out)
3:      for from ← 0 to neurons_per_layer[gap] pardo
4:          for to ← 0 to neurons_per_layer[gap+1] do
5:              out[to] ← 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 Prozesse desto langsamer.
- Vermutlich zu hoher Overhead durch Fork/Join bei wenig Iterationen.

Funktioniert das Training?

TODO:

Zahlen

TODO: Commits, LoCs, GitHub URI