Entwickung und Evaluation eines Fahrzeugsimulators für künstliche DNA

Bachelorarbeit von Dominik Mattern

2. August 2018

Betreuung: Prof. Dr. Brinkschulte und Dr. Pacher
Institut für Informatik
Goethe-Universität Frankfurt am Main

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Zusammenfassung

Es sollte eine Simulationsumgebung mit einem Straßennetz und eine KDNA, die Autos auf diesem Straßennetz kontrolliert, implementiert werden. Für die Simulation wurde eine einfache graphische Darstellung entwickelt auf der eine variable Anzahl Autos auf einem vorprogrammierten Straßennetz fahren. Eine KDNA steuert diese Autos über Kontrolle von Gas-, Bremse- und Steuerradpositionen, wobei Geschwindigkeits- und Richtungskontrolle unabhängig stattfinden. Bei der Analyse der KDNA für mehrere Autos traten Leistungsprobleme auf, deren Quelle genauer untersucht wurde. Die Last wurde primär durch die Kommunikation zur Verwaltung der KDNA-Tasks im AHS erzeugt.

2 Einführung

Beim Künstlichen Hormonsystem, zu Englisch Artificial Hormone System (AHS), handelt es sich um ein System, das Programmabläufe reguliert und auf heterogenen Prozessorsystem verwaltet[1]. Konzeptionell ist es an einem biologischen Hormonsystem angelehnt, daher der Name. Die Abläufe werden in der künstlichen DNA (KDNA), einer Datei aus verknüpften, vordefinierten Funktionsblöcken, sogenannten Tasks, definiert. Ausgeführt wird das System auf virtuellen Prozessorelementen.

2.1 Künstliches Hormon System

Die Organisation der Tasks auf den Prozessoren funktioniert über "Hormone". Dabei handelt es sich um Signale, die zwischen Prozessoren versandt werden und die Verteilung der Tasks auf den Prozessoren bestimmen. Dazu schüttet jeder Prozessor "Eagervalue" Hormone aus, die aussagen, wie sehr er selbst für einen Task geeignet ist. "Suppressoren" reduzieren diesen Eignungswert und werden durch Veränderungen der Last auf einem Prozessor verursacht. Nimmt ein Prozessor einen Task auf, sendet er außerdem Suppressoren an jeden anderen Prozessor, damit diese den Task nicht ebenfalls aufnehmen. "Accelerators" werden für gewisse Tasks an benachbarte Prozessoren ausgesendet und sorgen effektiv für Selbst-Optimierung. Sie werden für relevante Tasks an benachbarte Prozessoren gesendet und erhöhen den Eignungswert der Prozessoren für diese Tasks, wodurch Gruppen von untereinander stark verknüpften Tasks auch auf nahe verbundenen Prozessoren ausgeführt werden. Die KDNA muss natürlich irgendwo ausgeführt werden. Dazu dienen in dieser Simulation virtuelle Prozessor-Elemente, sogenannte "General Purpose Processors". Diese können KDNA laden, haben eine begrenzte Anzahl an Tasks, die sie ausführen können, und besitzen eine simulierte Last, die den Prozessor erhitzt. Falls Prozessoren versagen oder degradieren, sorgt die veränderte Hormonlage dazu, dass Tasks automatisch auf andere, bessere Prozessoren umgelagert werden.

2.2 Künstliche DNA

Die KDNA definiert das System, das vom AHS reguliert wird. Sie besteht aus Kombinationen aus Funktions-Bausteinen, sogenannten Tasks, mit Ein- und Ausgabekanälen, die beliebig verbunden werden können. Sensoren und

Aktoren stellen die Schnittstelle mit der Umgebung dar. Sensoren fragen Werte aus der Umgebung ab, Aktoren reichen die Werte, die sie von der KDNA erhalten, an die Umgebung weiter. Durch die Verknüpfung verschiedener Tasks untereinander wird so ein Prozess-Ablauf definiert, wobei eine Nachricht bei einem Quellen-Task wie einem Sensor startet, zu verarbeitenden Tasks weitergereicht und dabei verändert und zum Schluss an einen Aktor zur Ausgabe gesendet wird. Zusätzlich besitzen die meisten Tasks Konfigurationsoptionen, z.B. besitzt ein ALU-Task eine Option, die bestimmt, welche Operation die ALU ausführt, oder ein Sensor-Task besitzt -eine Option, die bestimmt, wie häufig er neue Werte ausliest. Eine KDNA-Datei ist dann eine menschenlesbare Textdatei, in der in jeder Zeile ein Task, die Verknüpfungen seiner Ausgangskanäle und eventuelle Optionen definiert werden.

In Abb. 1 ist ein einfacher Regelkreis dargestellt. Eine ALU vergleicht gemessene Temperaturwert mit einer Konstante, das Ergebniss wird an einen PID geschickt, dessen Output über einen Digital-Analog-Konverter zu einer Heizung geschickt werden.

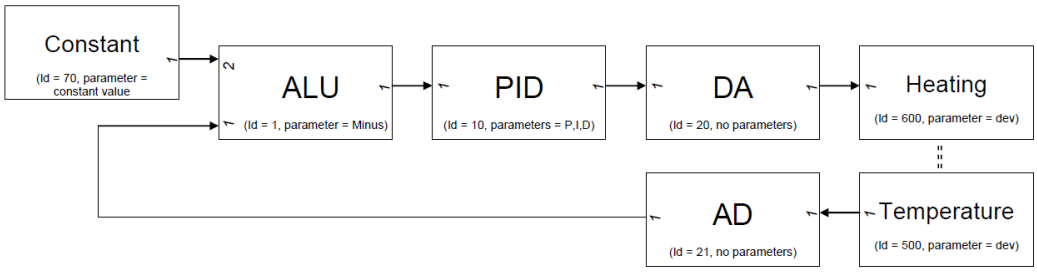

Abbildung 1: Einfacher Regelkreis zur Temperaturkontrolle. [1]

3 Entwicklung

3.1 Simulationsumgebung

Das Programm besteht aus einer Simulationsumgebung, die Nachrichten mit dem AHS austauscht. Da es eine Automobilsimulation sein soll, wurde ein Straßennetz als ungerichteter Graph implementiert. Jede Kante ist eine Straße, mit je einer Spur in beide Richtungen, jeder Knoten ist eine Kreuzung, die einfach als eine kreisförmige Fläche dargestellt wird. Ampeln oder besonderes Kreuzungsverhalten wurde nicht implementiert. Autos fahren an eine Kreuzung heran, halten an ihr, testen ob die Kreuzung leer ist, und falls ja, fahren dann über die Kreuzung zur Zielstraße. Ist die Kreuzung nicht leer, wartet das Auto, bis die Kreuzung leer ist. Die Zielstraße wird von einer Funktion geliefert, die Autosteuerung selbst weiß also nicht, wie es an der nächsten Kreuzung fahren soll, bis es diese erreicht hat. Sowohl das Straßennetz als auch die Straßen, die die Autos unterwegs wählen, sind fest vorprogrammiert.

Die Autos erhalten und senden Nachrichten zwischen der Kontrol-KDNA und der Simulationsumgebung über eine Reihe von Sensor- und Actor-Tasks. Die Sensor-Tasks können die Geschwindigkeit und die Richtung des Autos und die Richtung der Straße, auf der das Auto fährt, abfragen. Zusätzlich berechnet die Simulation die nach vorne freie Distanz entlang der Richtung der Straße und die Distanz zwischen dem Auto und der Mittellinie der Spur, auf der das Auto fahren sollte. Diese Werte können ebenfalls von der KDNA abgefragt werden. Dies könnte einem realen System entsprechen, in dem ein Sensorpaket rohe Daten von Kameras verarbeitet und daraus die entsprechenden Werte für das AHS bestimmt. Gesteuert wird das Auto durch die Actor-Tasks, die die Positionen von Gas, Bremse und Steuerrad direkt kontrollieren. Die Simulationsumgebung empfängt die Werte für Gas, Bremse und Steuerrad von der KDNA und verändert dann die Geschwindigkeit und Richtung des Autos entsprechend. Eine komplizierte Fahrphysiksimulation mit Impuls oder Reibung ist nicht implementiert, die Beschleunigung ist direkt proportional zur Position des Gaspedals oder der Bremse und einem festgesetzten maximalen Beschleunigungswert. Die Richtungsänderung hängt von Position des Steuerrads ab und beachtet, dass Autos sich nicht auf der Stelle drehen können und einen minimalen Wendekreis haben. Zur Steuerung benötigen die Autos den Abstand von der Spurmitte auf der sie fahren sollen und die Richtung der Straße. Dazu speichert jedes Auto sowohl die Straße, auf der es fährt, als auch die Kreuzung, an der die Straße endet.

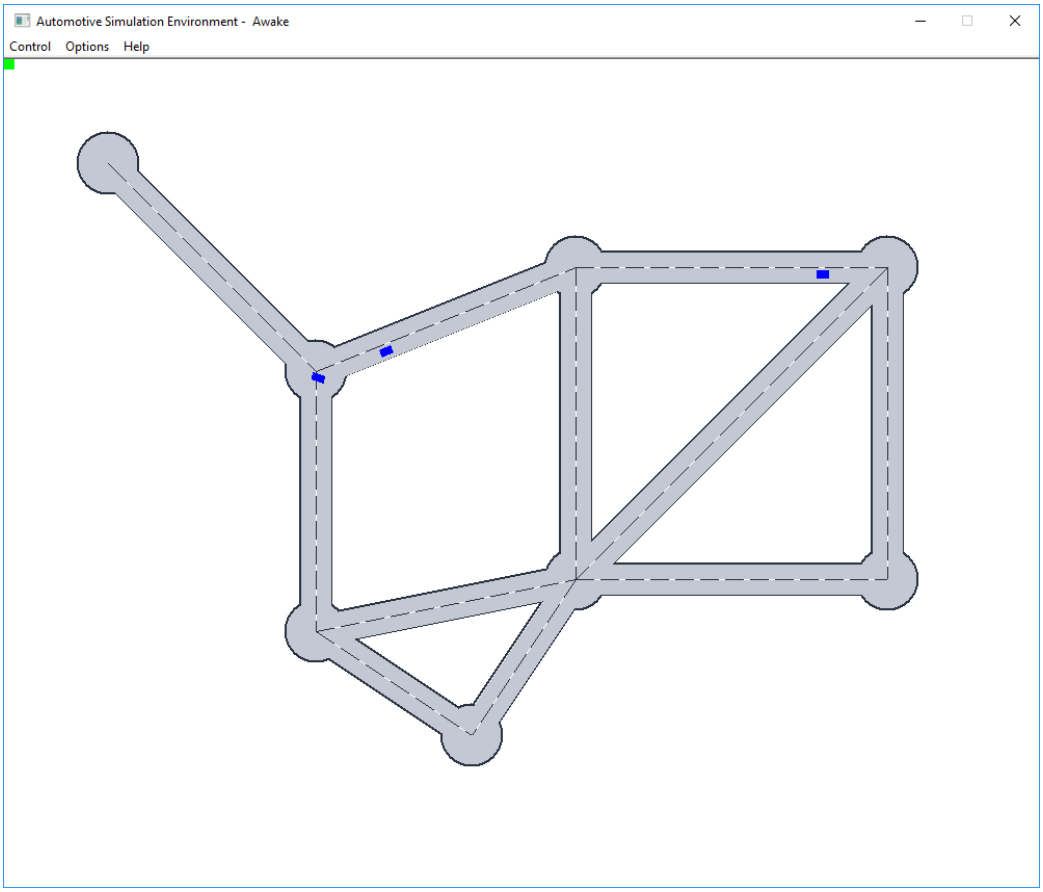

Abbildung 2: Screenshot der Simulationsumgebung. Die blauen Rechtecke stellen Autos dar, die grauen und schwarzen Streifen und Markierungen die Straßen

Erreicht das Auto eine Kreuzung und ist die Kreuzung frei, werden die Straße und Endkreuzung sofort geändert und die normale Spursteuerung bringt das Auto auf die neue Straße. Alle Autos sind identisch. In der graphischen Umgebung werden sie als blaue Rechtecke dargestellt.

Es findet keine Kollisionsüberprüfung statt, an sich könnte ein Auto also einfach von der Straße oder durch ein anderes Auto fahren. Dies hatte ungeplante Vorteile, da die Autos nun eine blaue Spur hinterlassen, wenn sie von der Straße abkommen, was es ermöglicht, das Verhalten der KDNA qualitativ zu untersuchen.

Der Simulationsprozess wurde als ein KDNA-Task implementiert, der automatisch beim Starten der Simulation auf einem dedizierten Prozessor ausgeführt wird. Zusätzlich zu diesem Simulationsprozessor wurde eine variable Anzahl "General Purpose Processors" gestartet, auf denen die tatsächliche Kontrol-KDNA ausgeführt wird. Dabei wird zwischen den KDNA-Prozessoren und dem Simulations-Prozessor, auf dem der Simulations-Task läuft, unterschieden, da diese unterschiedliches Verhalten aufweisen. Falls eine Anzahl Prozessoren ohne weitere Qualifikation angegeben wird, handelt es sich hier immer um KDNA-Prozessoren.

Optional kann der Zeitverlauf der Attribute der Autos wie Geschwindigkeit in eine Log-Datei geschrieben werden.

Das anfängliche Ziel der Arbeit, der Austausch von Tasks zwischen Autos nur unter bestimmten Bedingungen, stellte sich als praktisch schwer zu bewerkstelligen dar. Es wäre möglich, auf jedem Auto ein eigenes AHS laufen zu lassen, dann ist aber kein Task-Austausch zwischen den Autos möglich. Es müsste eine neue Struktur entwickelt werden, die über den einzelnen AHS läuft. Lässt man alle Autos auf einem AHS laufen, müsste man die virtuellen Prozessoren irgendwie den Autos fest zuweisen, und die Kommunikation zwischen Prozessoren so einschränken, das sie nur unter bestimmten Bedingungen möglich ist. Dies würde tiefe Eingriffe in die Struktur des AHS benötigen. Beide Methoden benötigten wesentlich mehr Zeit als für einen Bachelor verfügbar ist.

3.2 Steuerungs KDNA

Die Steuerungs-KDNA besteht aus zwei Bestandteilen, die getrennt die Geschwindigkeit und Richtung des Fahrzeugs kontrollieren. Beide funktionieren so, dass sie eine Soll-Richtung, bzw. Geschwindigkeit, bestimmen und dann das Auto auf diese Sollwerte zusteuern.

Die Geschwindigkeitskontrolle bestimmt eine maximale "sichere" Geschwindigkeit, so dass das Auto innerhalb der nach vorne freien Distanz bremsen könnte. Die tatsächliche Geschwindigkeit des Autos wird dann mit dieser sicheren Geschwindigkeit und der Geschwindigkeitsbegrenzung der Straße verglichen. Fährt das Auto langsamer als es sicher oder legal fahren könnte, beschleunigt es, ansonsten bremst es. Bremse und Gas werden nie zusammen betätigt. Die Geschwindigkeitssteuerung besteht effektiv aus drei Modulen, wobei das erste die Soll-Geschwindigkeit berechnet, das zweite den Vergleich mit zwischen Soll-Geschwindigkeit und der echten Geschwindigkeit bestimmt und das dritte Modul dann entsprechend bremst oder beschleunigt.

Die Richtungssteuerung berechnet aus der Distanz von der Mitte der Spur eine Ziel-Richtung, so dass das Auto zurück auf die Straße geführt wird. Der Unterschied zwischen der Ziel-Richtung und der tatsächlichen Richtung des Autos wird dann in einen PID geschleust, der das Steuerrad kontrolliert.

Das Auto ordentlich auf die Spur zurückzusteuern ist keine triviale Aufgabe. Zuerst wurde versucht, direkt die Abweichung von der Spur in einen PID einzuschleusen, der das Steuerrad kontrolliert. Es stellte sich aber als praktisch unmöglich heraus, für das gegebene Straßennetz einen Parametersatz für den PID zu bestimmen, der sowohl das Auto auf der Spur hält, als auch den Straßenwechsel an Kreuzungen ordentlich veranstaltet.

Der implementierte Ansatz berechnet stattdessen eine Soll-Richtung R_Z. Für sie gilt, dass sie identisch ist zur Straßenrichtung R_S, falls die Abweichung von der Spurlinie $d = 0$ ist. Ist die Abweichung nicht null, sorgt ein Modifier-Term R_M dazu, dass die Soll-Richtung zur Straße hinzeigt, also das Auto zurück zur Straße führt. Je größer die Abweichung, desto stärker steuert R_M auf die Straße zurück. Für R_Z gilt dann einfach: $R_Z = R_S + R_M$, wobei $R_M = \frac{d}{|d|+10} * \frac{\pi}{2}$ im Bereich $(-\frac{\pi}{2}, \frac{\pi}{2})$ liegt.

Von der Soll-Richtung wird nun die echte Fahrrichtung R_C abgezogen um das Steuersignal S zu erhalten. Ist dieses Signal negativ, steuert das Auto nach links, ist es positiv, steuert das Auto nach rechts. Nun kann es aber an der 0 Grenze zu einem Problem kommen. Liegt die Soll-Richtung gerade über 0, die tatsächliche Fahrrichtung aber gerade darunter, also bei nahe 2π,

ist das Signal stark negativ und das Auto steuert über den "langen" Weg, siehe Abb. 3.2.

Dies führt dazu, dass das Auto einen großen Bogen schlägt. Dabei ändert sich nun aber auch der Abstand von der Spurlinie und damit die Soll-Richtung und es kann dazu kommen, dass das Auto nie die momentane Soll-Richtung erreicht und ewig in einem Kreis fährt. Das Steuersignal wird korrigiert, indem man die Winkel auf den Bereich $(-\pi, \pi)$ umrechnet, indem man von Werten die oberhalb dieses Bereiches 2π abzieht und auf Werte unterhalb 2π drauf rechnet:

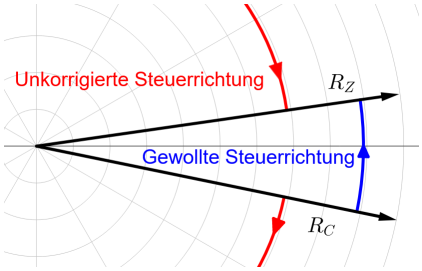

Abbildung 3: Korrektur des Steuerrungssignals

$$S_{korr} = \begin{cases} S + 2\pi, & \text{if } S < -\pi \\ S - 2\pi, & \text{if } S > \pi \\ S, & \text{sonst} \end{cases} \tag{1}$$

Das korrigierte Steuersignal S_{korr} wird nun an einen PID weitergegeben, dessen Ausgabesignal nochmal normiert wird und dann über den Aktor-Task die Steuerradposition des Autos bestimmt.

Flowcharts die die Tasks der Geschwindigkeits- und Richtungssteuerung der KDNA für ein Auto darstellen sind in Abb. 4 und 5 dargestellt.

Zusätzlich ist ein KDNA-Checker Task als Notstopp implementiert. Dieser besteht aus einem KDNA-Checker Task und einem Actor, der die Ausgabe des Checker-Tasks an die Simulation weiterreicht. Wird die KDNA nicht mehr komplett ausgeführt, oder wird von dem Checker Task eine gewisse Zeit lang keine Nachricht erhalten, wird ein Sicherheitsprotokoll aktiviert, dass die Autos stoppt. Dadurch werden die Autos zu einem Stopp gebracht, auch wenn die Kontrolaktoren oder der Checker-Task selbst nicht ausgeführt werden. Ob der Notstopp gerade aktiv ist oder nicht, wird durch ein rotes oder grünes Rechteck in der oberen linke Ecke des Fensters angezeigt, wobei Grün bedeutet, dass der Stopp nicht aktiv ist und die Autos von der KDNA kontrolliert werden.

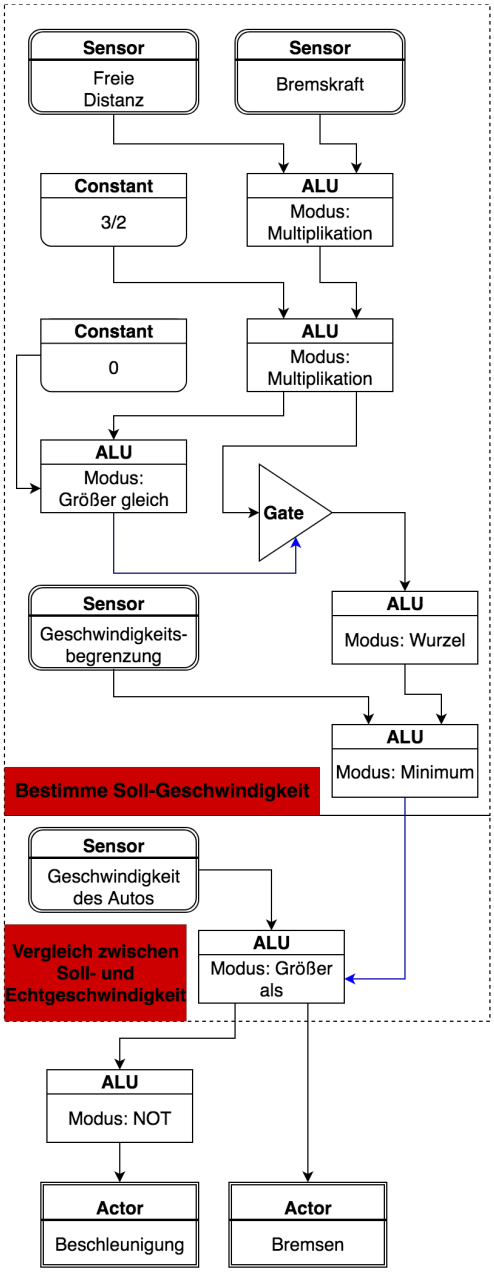

Abbildung 4: Flowchart der KDNA-Tasks, die die Geschwindigkeitskontrolle eines einzelnen Autos betreiben. Bei Operationen, deren Operanden nicht austauschbar sind, oder bei Steuersignalen ist das zweite Eingangssignal, bzw. Steuersignal blau gefärbt.

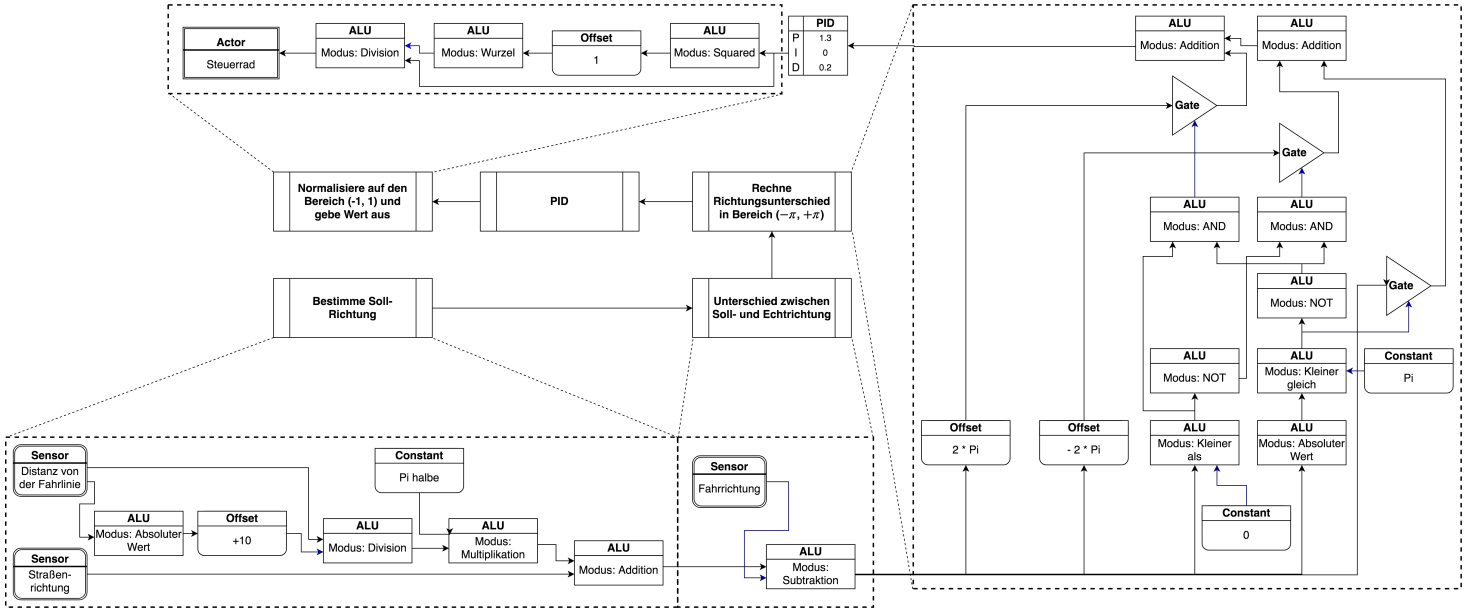

Abbildung 5: Flowchart der KDNA-Tasks die die Richtungskontrolle eines Autos betreiben. Bei Operationen, deren Operanden nicht austauschbar sind, oder bei Steuersignalen ist das zweite Eingangssignal, bzw. Steuersignal blau gefärbt.

Während der Entwicklung der KDNA traten mehrere praktische Probleme auf. Werden nicht alle KDNA Tasks ausgeführt, kann es zu unerwartetem Verhalten kommen, z.B. bremst das Auto nicht mehr, falls der Bremsaktor nicht ausgeführt wird. Da Tasks nicht priorisiert werden können, muss fundamentale Sicherheit außerhalb der Simulation laufen. Es gibt zwar einen KDNA-Checkter Task, der die Ausführung der KDNA überprüft und ein Totsignal schickt, falls dies nicht der Fall ist, aber der Checker-Task selbst kann unsauber abbrechen, ohne sein letztes Signal zu senden. Dieses Problem wurde gelöst, indem zusätzlich eine Uhr läuft, die regelmäßig zurückgesetzt wird, solange der Checker-Task läuft. Wird der Checker-Task nicht mehr ausgeführt, wird die Uhr nicht mehr zurückgesetzt und sobald die Uhr einen gewissen Wert überschreitet, werden alle Autos gestoppt. Die Uhr und die Stoppfunktion müssen dabei außerhalb der KDNA implementiert werden.

Ein fundamentales Prinzip der KDNA ist es, dass Tasks Eingabesignale erwarten und ein Mangel an Eingabesignalen ein Problem mit dem sendenden Task anzeigt. Dies führt unter anderem dazu, dass Gates immer einen Wert weitergeben, entweder das Eingabesignal oder ein Nullsignal. Kontrollstrukturen benötigen dann immer eine relativ umständige Struktur aus Gates und kontrollierenden ALUs. Auch gibt es in den existierenden Task Definitionen keine mit variabler Anzahl an Eingangssignalen, man muss also mehrere ALUs verketten, um z.B. von mehr als zwei Termen das logische UND zu bilden. Es ist auch kaum möglich, Tasks "wiederzuverwenden", denn jeder Eingangskanal kann vor nur genau einem Task Signale empfangen. Ein Multiplexer besteht immer aus zwei Eingängen und einem Ein-Bit-Steuereingang, größere Multiplexer existieren nicht und müssen aus mehreren 2-Multiplexern und ALUs konstruiert werden.

Eine Konsequenz daraus ist, dass schon relativ einfache Programmabläufe eine größe Anzahl an Tasks benötigen, die nun untereinander kommunizieren und vom AHS verwaltet werden müssen. So braucht die Richtungsnormierung in der Steuerung, in Abb. 5 zu sehen, 16 Tasks und macht damit die Hälfte der gesamten Richtungssteuerung aus, obwohl es eigentlich ein relativ einfaches if-else Konstrukt ist. Insgesamt besteht die KDNA aus 47 Tasks pro Auto zur Kontrolle und 2 Tasks für den Notstopp. Die KDNA für ein Auto ist im Anhang aufgelistet.

Da mehrere Autos vom selben AHS kontrolliert werden, müssen die KDNA-Tasks für die verschiedenen Autos in die gleiche Datei geschrieben werden. Es muss also eine neue KDNA-Datei her, je nachdem wie viele Autos existieren.

4 Evaluation

4.1 Verhalten der Simulation

Für ein einzelnes Auto mit der minimalen Anzahl an nötigen Prozessoren verhielt sich das System wie erwartet. Das Auto beschleunigt zwischen Kreuzung bis zur Geschwindigkeitsbegrenzung, bremst vor der Kreuzung, so dass es gerade an der Kreuzung zum Stopp kommt, und fährt dann über die Kreuzung zur nächsten Straße weiter. Wird einer der Prozessoren abgeschaltet, kickt der Sicherheitsstopp ein und stoppt das Auto. Bei sehr scharfen Kurven fährt das Auto über die Straßengrenze, insbesondere gibt es kein Verhalten bei solchen Kurven erst in die andere Richtung auszuschlagen. Auch bei Wendemanouvern kann das Auto nicht auf der Straße bleiben.

Erhöht man die Anzahl Prozessoren scheint das Systemverhalten sich zuerst nicht zu ändern, so ist auch für 15 Prozessoren kaum Abweichung von dem 4 Prozessor-verlauf zu sehen. Bei 20 Prozessoren kann das Auto nicht mehr effektiv die Spur halten und bei 22 war der Gastrechner so überlastet, dass das Programm häufig stockte und es fast 30 Sekunden dauerte, bis das Auto überhaupt anfing zu fahren. Dieses Verhalten kann sowohl aus der graphischen Umgebung, siehe Abb. 6, als aus den geloggten Werten abgelesen werden.

In Abb. 7 und 8 ist der zeitliche Verlauf für die Geschwindigkeit und Spurabweichung aus der Log-Datei dargestellt. Für 4 Prozessoren sieht man, wie das Auto zur Geschwindigkeitsgrenze beschleunigt, dann vor Kreuzung bis fast zum Stop abbremst, über die Kreuzung fährt und wieder beschleunigt. Die tiefen Einkerbungen stellen also Kreuzungen dar. Für 20 und 22 Prozessoren sieht man klar wie irregulär die Beschleunigung ist. Das Auto bremst zwischendurch regelmäßig, entweder wegen Fehlverhalten der KDNA oder dem Sicherheitsstopp, und es beschleunigt auch weit über die Geschwindigkeitsbegrenzung hinaus. Dass das Auto nicht nur irregulär bremst zeigt klar, dass die KDNA fehlerhaft ausgeführt wird, da dies sonst nicht möglich sein sollte. Da die durchschnittliche Geschwindigkeit niedriger ist, wird die erste Kreuzung später erreicht. Man sieht auch, dass der Sicherheitsstopp vielleicht das Auto zwischendurch kurz abbremst, aber es nicht zum Anhalten bringt. Die verzögerte Beschleunigung am Start wird durch die Zeitverzögerung zwischen Start des Loggings und der Simulation ausgelöst und hat keine weitere Bedeutung. Man kann auch sehen, dass die Durchläufe anscheinend unterschiedliche Dauer hatten. So scheint der Durchlauf mit 22 Prozessoren schon

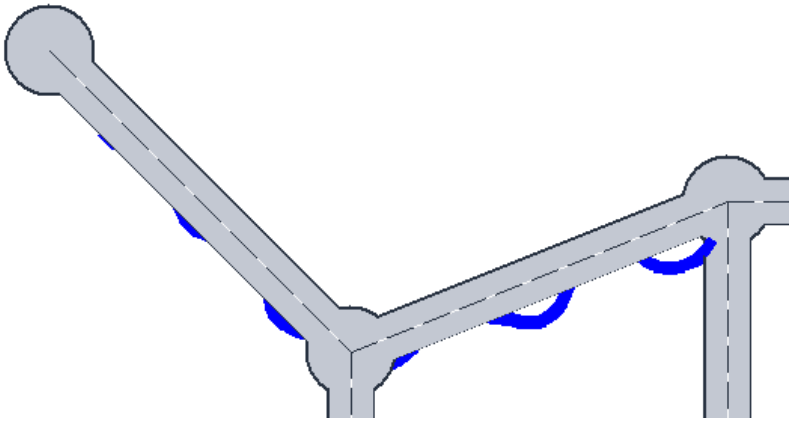

Abbildung 6: Ein Bild der Simulation nach einer Minute Laufzeit auf 20 Prozessoren. Die blauen Streifen zeigen an, wo das Auto über den Straßenrand hinausgetreten ist. Das Auto selbst ist ganz rechts dabei, auf die Kreuzung zu fahren.

nach ungefähr 35 Sekunden abzubrechen. Der Unterschied entsteht dadurch, dass die dargestellte Zeit die Simulationszeit ist. Ist die Simulation also eingefroren, z.B. beim Aufwachen des Systems, vergeht keine Simulationszeit und es werden keine Werte geloggt. Die Simulation wurde aber für 60 Sekunden Realzeit laufen gelassen. Aus dem Unterschied kann man daher grob ablesen, wie viel Zeit bei Stockungen der Simulation verloren ging.

Auch bei der Spurabweichung sind die Kreuzungen klar zu erkennen, da beim Straßenwechsel die Straße auf der sich das Auto "befindet" spontan gewechselt wird und die Abweichung der Spur sich dann auch schlagartig ändert. Für 4 Prozessoren pendelt das Auto sich relativ schnell ein und bleibt dann zentral auf der Spur. Bei 20 Prozessoren hingegen pendelt es wild von links nach rechts und fährt überhaupt nur im Durchschnitt entlang der Straße. Für 22 Prozessoren schlägt das Auto noch weiter aus.

Offensichtlich führt eine Überlastung des Systems dazu, dass das Verhalten sich fundamental ändert, die Simulation wird nicht nur langsamer ausgeführt. Der Notstopp war zwischendurch immer wieder kurzzeitig mal aktiv, unterband das fehlerhafte Verhalten aber nicht. Das Konstrukt aus KDNA-Checker Task und Timer ist also nicht genug, um komplette Sicherheit zu garantieren.

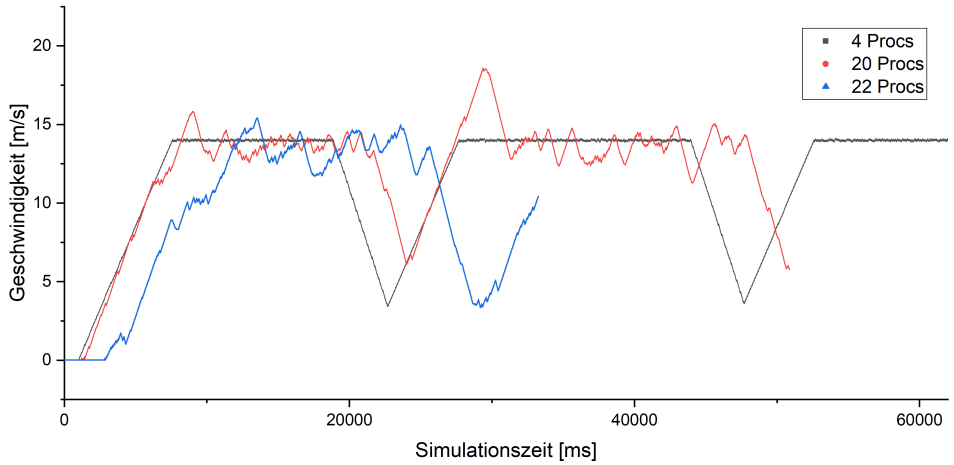

Abbildung 7: Zeitverlauf der Geschwindigkeit des Autos für 4, 20 und 22 Prozessoren. Die Zeit ist in Simulationszeit, friert die Simulation ein vergeht also keine Zeit.

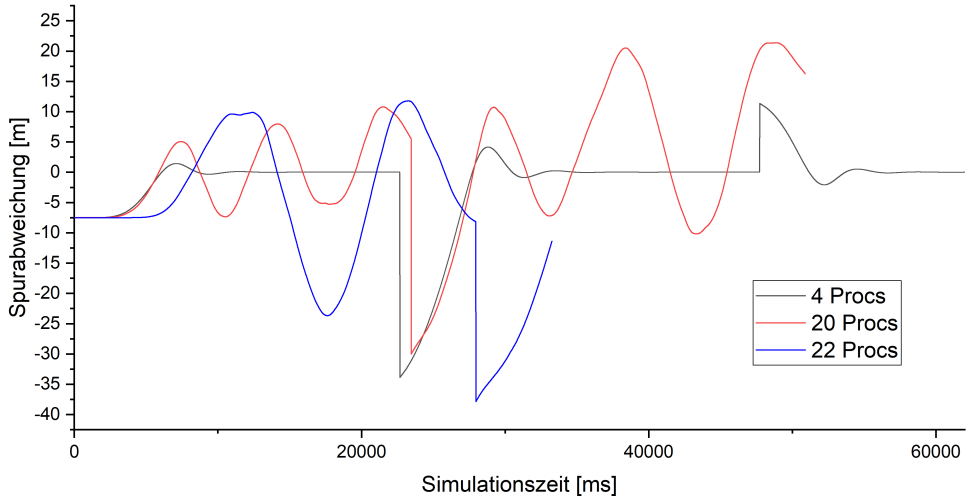

Abbildung 8: Zeitverlauf der Abweichung von der Spurmittellinie für 4, 20 und 22 Prozessoren.

Eine mögliche Ursache für das Fehlverhalten ist, dass es keine Garantie gibt, dass die Tasks in einer KDNA synchron ausgeführt werden. Es ist also möglich, dass ein hinterer Task mit alten Werten eines Vordertasks weiter rechnet, bevor der Vordertask nochmal ausgeführt wird. Es gibt auch keine Garantie, dass die Anzahl Aufrufe zueinander passen. Es ist also möglich, dass ein hinterer Task mehrmals mit demselben alten Wert rechnet, bevor der Vordertask ausgeführt wird. Bei geringer Last ist die Häufigkeit und Dauer dieser kurzen Aussetzer klein genug, dass das System als ganzes sich wie erwartet verhält. Bei hoher Auslastung ist dies nicht mehr der Fall und gewisse Tasks werden temporär effektiv nicht ausgeführt, auch wenn sie offiziel noch aktiv sind.

Ein weiterer möglicher Grund ist die stark erhöhte Anzahl an Taskumlagerungen zwischen Prozessoren. Die Tasks werden während der Umlagerung effektiv nicht ausgeführt, und bei hoher Auslastung könnte die Umlagerung längere Zeit in Anspruch nehmen. Auch hier ist keine zeitlichen Konsistenz garantiert. Da das Verhalten bei geringer Auslastung wie erwartet ist, die KDNA die Autos also effektiv steuern kann, ist es bei dieser Anwendung kein Problem wenn gelegentlich ein paar Werte etwas veraltet sind. Eine genauere Analyse der Zeitdauer und ihrer Varianz zwischen Ausführung des selben Tasks wurde hier nicht durchgeführt.

Erhöht man die Anzahl Autos auf den Straßen, muss auch die KDNA entsprechend vergrößert werden, bei 3 Autos beinhaltet die KDNA also dreimal soviele Tasks. Entsprechend höher ist die Last auf dem System und der Gastrechner war schon bei der minimalen Anzahl Prozessoren, 11, nahe der Leistungsgrenze. Die Geschwindigkeiten für alle drei Autos sind in Abb. 9 dargestellt. Interessant ist, wie ähnlich die Geschwindigkeitskurven für 13 und 15 Prozessoren zwischen allen Autos ist. Dieses Verhalten tritt aufgrund des Notstops auf, der nur für die gesamte KDNA wirkt, und daher alle drei Autos identisch stoppt. Für die größere KDNA war der Notstop also wesentlich effektiver darin, das fehlerhafte Verhalten zu unterbinden.

In allen Fällen kam es bei einer hoher Zahl Prozessoren dazu, dass die Simulation komplett einfror und nicht mehr reagierte.

Da die Last unter anderem von der Anzahl an Tasks abhängt, wurde versucht ein Teil der KDNA, der Werte verrechnet, in die Simulation umzulagern. Dazu wurde das Richtungsverrechnungsmodul in der Richtungskontrolle durch einen Aktor und einen Sensor ersetzt. Der Aktor gibt den Eingabewert an die Simulation weiter, die diesen Wert speichert. Wenn der Sensor dann eine Anfrage stellt, liest die Simulation den gespeicherten Wert

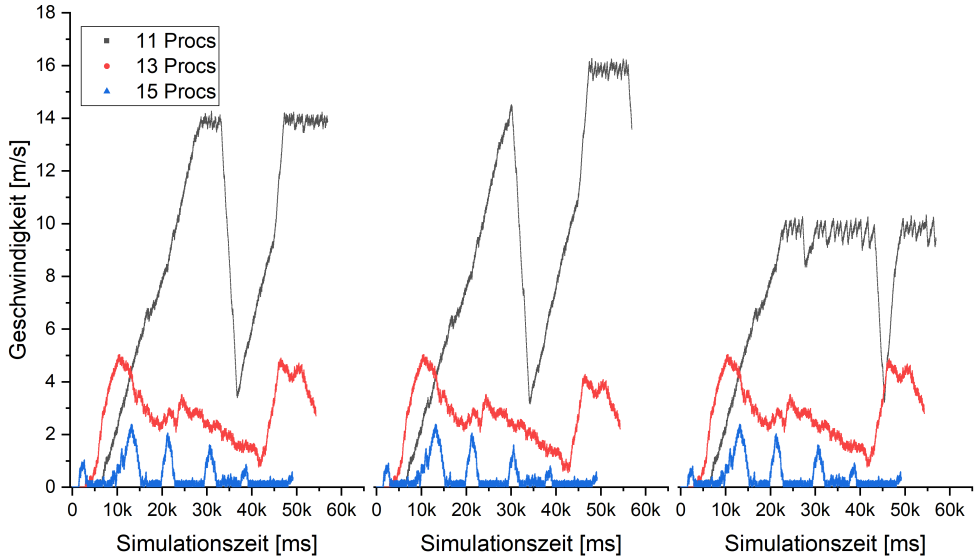

Abbildung 9: Geschwindigkeitsverlauf der drei Autos für verschiedene Prozessorzahlen.

aus, macht die nötige Rechnung und gibt den Ausgabewert an den Sensor weiter, der den Wert dann an den Rest der KDNA weiterreicht, siehe Abb. 10. Da keine synchrone Ausführung garantiert ist, muss der Ausgabewert initialisiert werden, falls die erste Sensoranfrage vor der ersten Aktorausführung stattfindet.

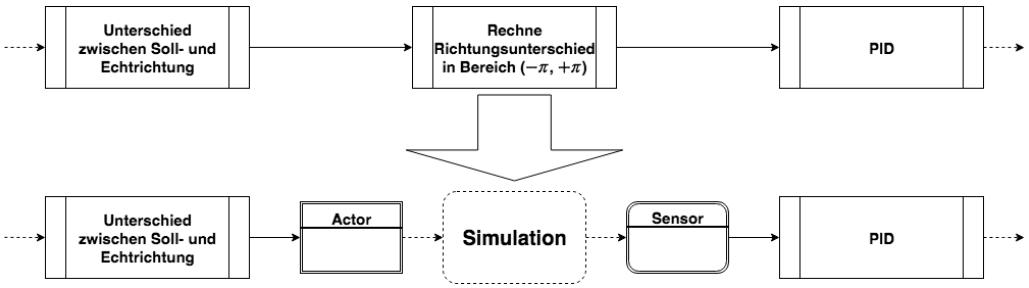

Abbildung 10: Illustration des geänderten Programverlaufes.

Da die Anzahl Tasks und damit auch die minimale Anzahl benötigter Prozessoren reduziert wird, sollte dies zu einer klaren Leistungsverbesserung führen. Für ein Auto ist dies auch klar bemerkbar, für 22 Prozessoren ist

die Simulation mit der kompakteren KDNA wesentlich stabiler. Im Vergleich zum normalen Verlauf dauert es aber wesentlich länger, bis die Simulation sich komplett eingependelt hat.

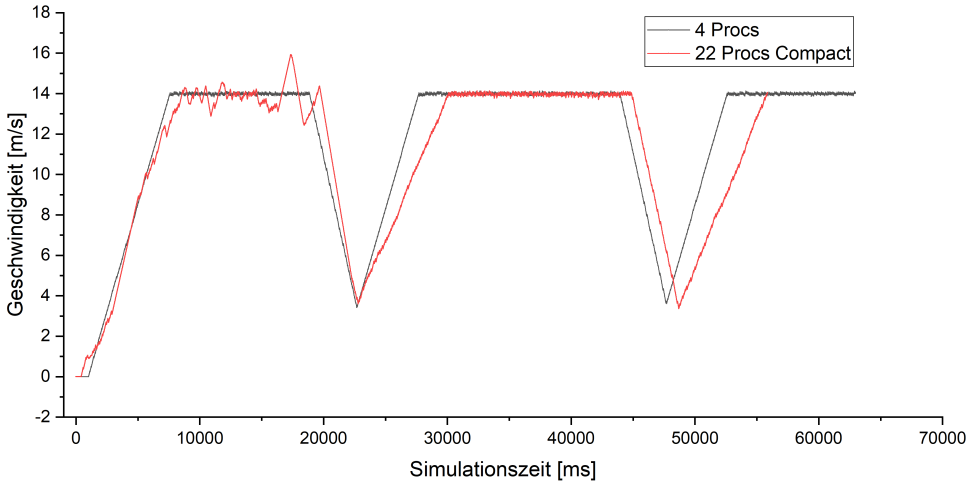

Abbildung 11: Vergleich des Geschwindigkeitsverlaufs für 4 Prozessoren mit der normalen KDNA und 22 Prozessoren mit der kompakten KDNA.

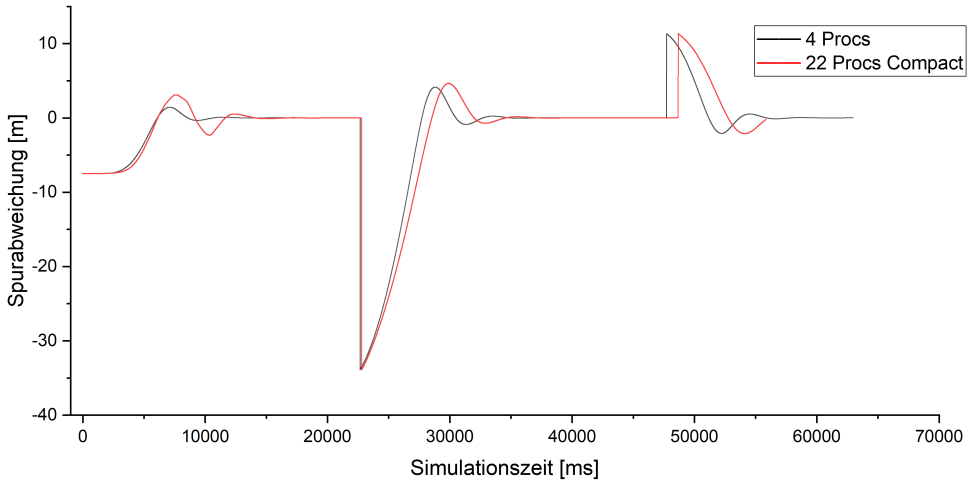

Abbildung 12: Vergleich der Spurabgleichung für 4 Prozessoren mit der normalen KDNA und 22 Prozessoren mit der kompakten KDNA.

4.2 Leistungsanalyse

Das System wurde für ein und drei Autos untersucht. Dazu wurde die Simulation jeweils eine Minute lang mit verschiedenen Prozessorzahlen laufen gelassen. Die Anzahl und Dichte der gesendeten Telegramme wurde dann aus den Log-Dateien genauer analysiert. Das AHS und die Prozessoren besitzen verschiedene Logstufen. Die höchste Stufe notiert dabei für jedes Telegramm, das über den entsprechenden Prozessor versandt wird, egal ob zwischen Tasks, Prozessoren oder ob es sich um Fehlernachrichten des AHS selbst handelt, den Timestamp und die Art des Telegrammes in einem Logfile. Schickt also ein Task A auf Prozessor X ein Ergebnis an einen anderen Task B auf Prozessor Y weiter, wird im Logfile von X notiert, dass eine Nachricht von A zu B auf Y geschickt wurde, und im Logfile von Y, dass eine Nachricht von A zu B empfangen wurde. Der Inhalt und verschiedene andere Information werden ebenfalls gespeichert.

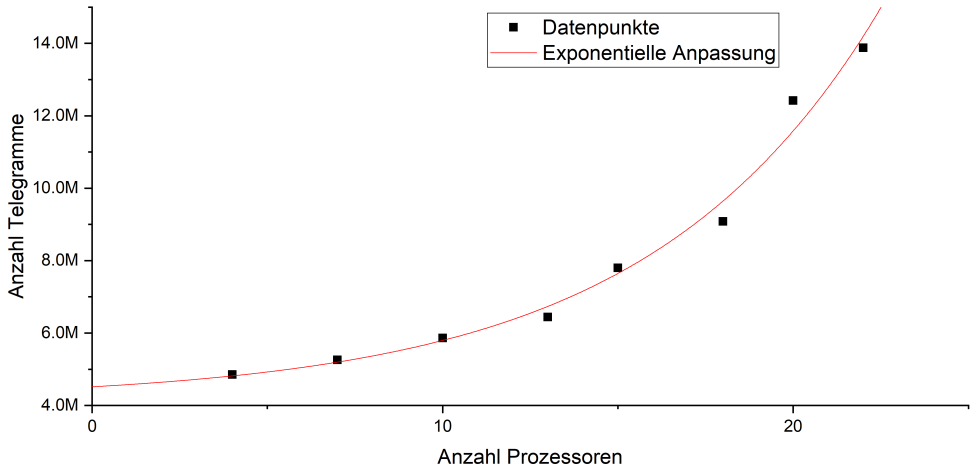

Abbildung 13: Anzahl aller gesendeten Telegramme für verschiedene Prozessorzahlen. Beinhaltet alle Telegramme der Simulations- und KDNA-Prozessoren.

Schon bei nur einem Auto traten für hohe Prozessorenzahlen klare Leistungsprobleme auf. Mindestens 4 Prozessoren waren nötig, um die KDNA komplett ausführen zu können. Auf 40 oder 30 Prozessoren fror die Simulation schon beim Aufwachen ein und der Computer musste neugestartet

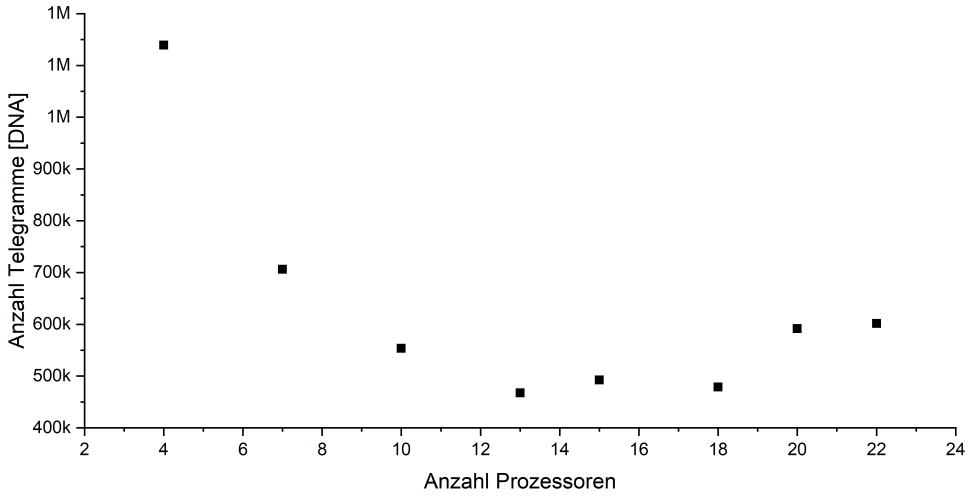

Abbildung 14: Graph der durchschnittlichen Anzahl gesendeter Telegramme der KDNA-Prozessoren pro Prozessor für verschiedene Prozessorzahlen.

werden. Die höchste getestete Anzahl Prozessoren, bei der die Simulation noch Lebenszeichen von sich gab, war 22. Man sieht in Abb. 13 klar, wie die Gesamtanzahl Telegramme bei steigender Prozessorzahl ebenfalls steigt. Dabei scheint der Anstieg nicht linear sondern quadratisch in der Prozessorzahl zu sein. In Abb. 14 ist die durchschnittliche Anzahl der Telegramme pro KDNA-Prozessor dargestellt. Diese sinkt zuerst mit steigender Prozessorzahl, allerdings nicht proportional dazu, und für ausreichend große Werte steigt sie sogar.

Exemplarisch ist in Tabelle 1 ein Teil der Log-Information für 4 KDNA-Prozessoren und den Simulationsprozessor aufgeführt. Aus Platzgründen sind einige Einträge entfernt worden. Manche Ereignisse, wie das Verschieben eines Tasks, verursachen pro Ereigniss mehrere verschiedene Einträge, von denen die meisten nicht aussagekräftig sind. Außerdem wurden Einträge für manche Telegramme entfernt, wie die Hormon Signale, die zum Start und Stopp der Simulation gesendet werden, da diese jeweils genau einmal ausgegeben werden.

Man sieht klar den Unterschied zwischen Simulations- und KDNA-Prozessoren. Der Simulationsprozessor verarbeitet keine KDNA-Nachrichten und die Hormonsignale der KDNA-Prozessoren werden vom Simulationsprozes-

Telegramme	KDNA-Prozessor 1	KDNA-Prozessor 2	KDNA-Prozessor 3	KDNA-Prozessor 4	Simulationsprozessor
Lifesign given from: Task	249683	264398	675646	442398	74965
KDNA message sent from task to task	86695	98462	195148	130703	0
KDNA parameter for task retrieved	22	27	27	18	0
Getting sensor data from task port	1901	4760	2056	3795	0
Sending actor data from task to port	577	9491	34117	0	0
Message received for task from processor task	69502	54976	239870	158633	56696
Message telegram received	23217	14430	80417	80266	56696
Message send: From task to task	89172	112707	231321	134498	12512
Message telegram sent to processor	42887	72161	71337	56129	12512
Relevant hormone received: Accelerator	36863	40907	35616	31896	3
Relevant hormone received: Eagervalue	12393	12393	12393	12393	9
Relevant hormone received: Suppressor	77542	77726	76416	75134	2520
Irrelevant hormone received: Eagervalue	5	5	5	5	12392
Irrelevant hormone received: Suppressor	2517	2517	2517	2517	61364
Sending load hormone to myself: Suppressor	16173	16359	15048	13766	2
Sending offer hormone to myself: Accelerator	671	683	627	626	2
Sending organ hormone to neighbors: Accelerator	36191	40223	34988	31269	0
Sending task hormone to all: Eagervalue	3048	3098	3147	3099	7
Sending task hormone to all: Suppressor	15862	16038	14758	14718	2517
Hormone telegram received: Length bytes	5045	5044	5044	5045	5046
Hormone telegram sent to all: Length bytes	1261	1262	1262	1261	1260
New location of task added: Task at processor	55	55	55	55	55
Errorcode occured	1	7	562	3	0
KDNA error occured	0	5	0	0	0

Tabelle 1: Log-Daten der 5 Prozessoren. Manche Einträge wurden verkürzt um Platz zu sparen. Die Daten wurden gesammelt, indem Zeitstempel und Zahleneinträge aus den Einträgen entfernt wurden, um die Einträge auf Nachrichtentypen zu reduzieren.

sor als irrelevant betrachtet und umgekehrt. Die irrelevanten Hormonsignale werden aber trotzdem noch empfangen und versendet. Man sieht auch, dass die Taskanzahl nicht unbedingt gleichmäßig zwischen den Prozessoren verteilt ist, Prozessor 3 hat wesentlich mehr Nachrichten empfangen und versandt und auch die Anzahl an Task-Lifesigns ist wesentlich höher. Die Anzahl Hormon-Nachrichten ist zwischen den Prozessoren aber trotzdem grob konstant. Außerdem sind einige Fehler aufgetreten, die aber keinen kritischen Einfluss auf den Verlauf der Simulation hatten.

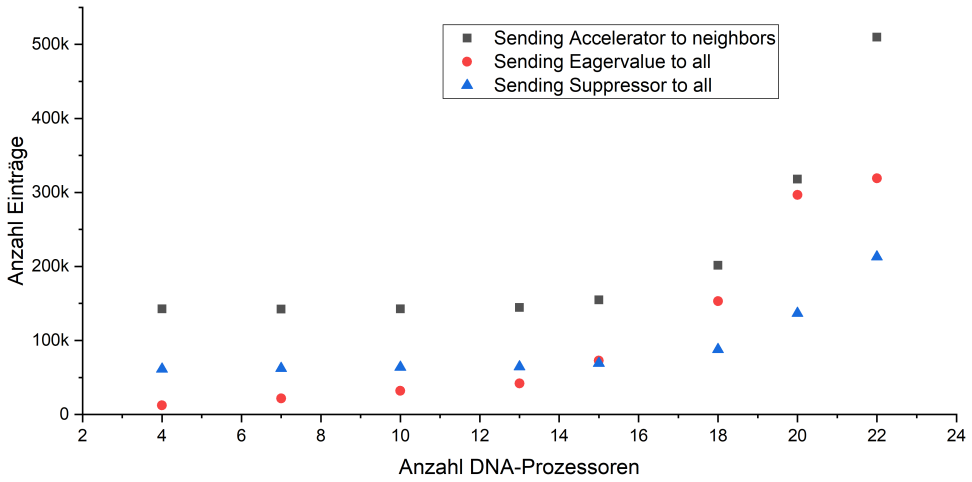

Abbildung 15: Graph der Gesamtanzahl an Log Einträgen für die entsprechenden Telegramme.

Die Anzahl gesendeter Hormonsignale ist stark von der Anzahl der Prozessoren abhängig. Die Anzahl Eagervalue Hormone, die ausgeschüttet werden muss, ist direkt proportional zur Prozessorzahl, da jeder Prozessor für jeden Task einen Eagervalue ausgeben muss. Eagervalues werden an alle Prozessoren versendet, die Anzahl Nachrichten ist also quadratisch in der Prozessorzahl. Suppressoren werden nur für aufgenommene Tasks ausgeschüttet, die ausgeschüttete Anzahl sollte also konstant sein. Suppressoren werden ebenfalls an alle Prozessoren verschickt, die Anzahl Nachrichten sollte also proportional zur Prozessorzahl sein. Accelerators werden ebenfalls nur für aufgenommene Tasks ausgeschüttet, allerdings nur an relevante Prozessoren, die Anzahl Nachrichten sollte also konstant sein. Die entsprechenden Einträge sind in Abb. 15 und 16 dargestellt. Tatsächlich verhalten sich die

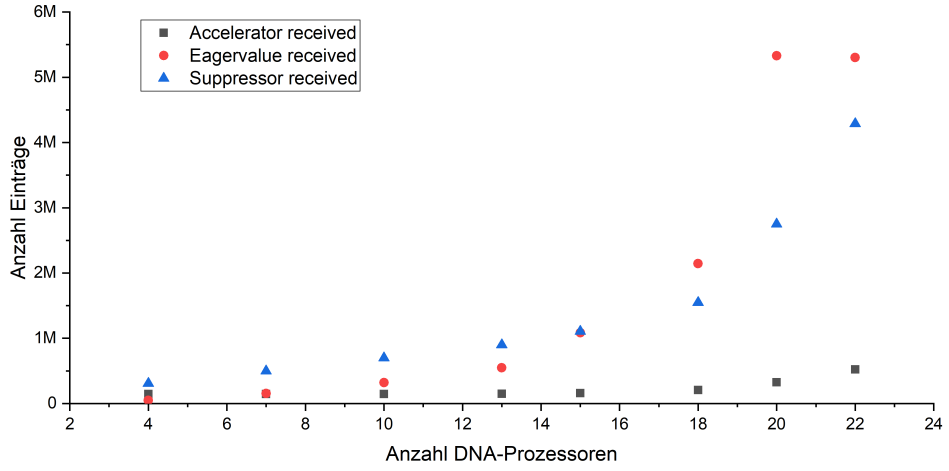

Abbildung 16: Graph der Gesamtanzahl an Log Einträgen für die entsprechenden Telegramme.

Anzahl Prozessoren	4	7	10	13	15	18	20	22
Errorcode occured	573	206	812	656	13999	25344	159903	106573
KDNA-Error occured	5	9	3	1	0	0	8	1

Tabelle 2: Anzahl Einträge für KDNA und sonstige Fehlermeldungen für verschiedene Prozessorzahlen

Anzahl gesendeter und empfangener Hormone zwischen 4 und 15 Prozessoren wie erwartet, steigen dann aber stark an.

Betrachten wir nun die KDNA bezogenen Einträge. Die Anzahl an gesendeten und empfangenen Nachrichten ist im fehlerfreien Verlauf der Simulation annähernd konstant, bricht dann aber bei 20 und 22 Prozessoren ein, siehe Abb. 17. Interessant ist, dass die Sensor-Abfragen nicht annähernd so stark zurückgehen, die Actor-Ausgaben aber doch, siehe Abb. 18. KDNA-Fehler treten nicht gehäuft auf, sonstige Fehler aber schon, siehe Tabelle 2, was zeigt, dass entweder die Diagnose der KDNA ein durch Überlastung erzeugtes Fehlverhalten nicht erkennt, oder dass das Fehlverhalten nicht durch Fehler in der Ausführung der KDNA, sondern in der Verwaltung der KDNA erzeugt wird. Die Anzahl an Taskumlagerung ist für geringere Prozessorzahlen beinahe konstant, steigt dann aber drastisch an, siehe Abb. 19.

In den anderen Einträgen, die sich mit Umlagerungen befassen, siehe

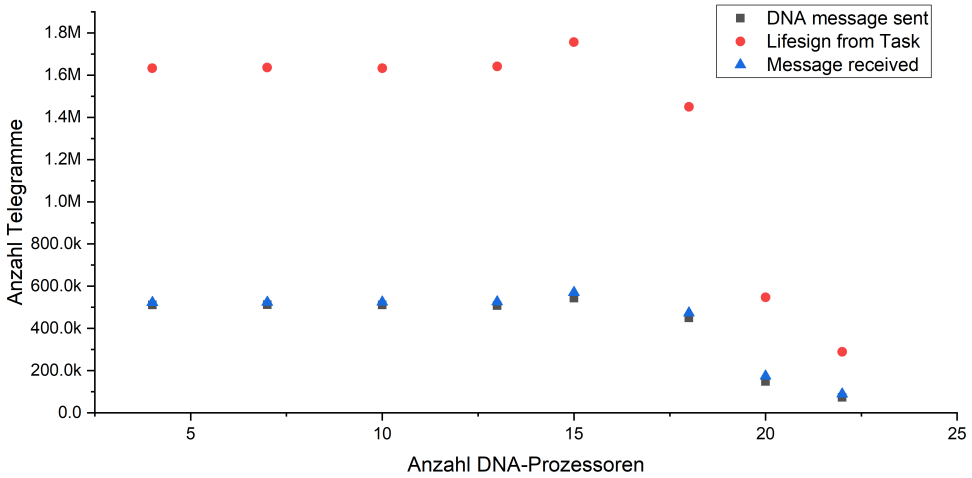

Abbildung 17: Graph der Gesamtanzahl an Log Einträgen für die entsprechenden Telegramme.

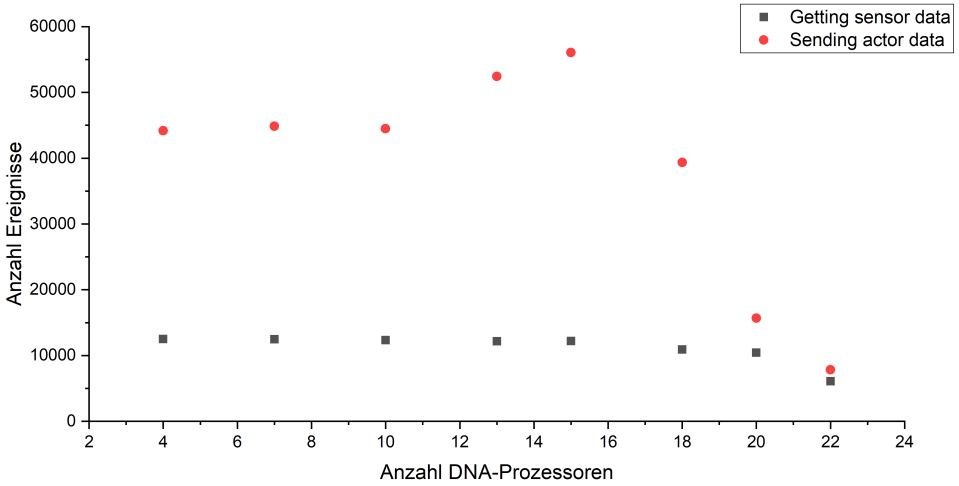

Abbildung 18: Anzahl der Sensor Abfragen und Actor Ausgaben für verschiedene Prozessorzahlen

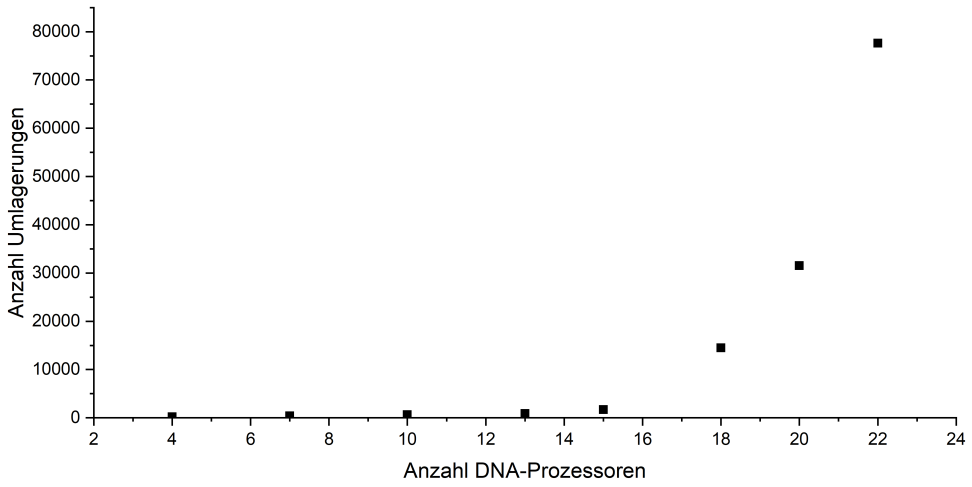

Abbildung 19: Anzahl der Taskumlagerungen für verschiedene Prozessorzahlen

Abb.20, sieht man, dass die Anzahl möglicher Umlagerung zuerst grob proportional zur Prozessorzahl ansteigt, diese aber meistens abgewiesen werden. Für 18, 20 und 22 Prozessoren werden die meisten Umlagerung nicht mehr zurückgewiesen. Interessant ist, warum diese Umlagerungen nicht mehr abgewiesen werden. Es ist möglich, dass es zu einem kaskadenartigen Versagen kommt, eine Veränderung der Last führt dazu, dass ein Task von einem Prozessor auf einen anderen umgelagert wird. Diese Umlagerung erhöht nun die Last auf den Rechner und verursacht dadurch weitere Umlagerung. Solange die langfristige Durchschnittslast klein genug ist, findet das System schließlich ein Equilibrium und läuft "normal" weiter.

Die Umlagerungen werden hier möglicherweise sowohl direkt durch die Last auf den Rechner als auch durch die Überlastung der Kommunikation ausgelöst. Ein Prozessor ist überlastet und versucht Tasks loszuwerden und sieht, dass ein anderer Prozessor diese möglicherweise besser ausführen könnte und hört auf, den Task auszuführen. Der Task muss nun auf einem anderen Prozessor ausgeführt werden, der nun ebenfalls versucht, Tasks zu verschieben, und die Verschiebungen und damit verbundenen Hormon- und Verwaltungsnachrichten tragen zur Kommunikationslast bei und das System stabilisiert sich nie. Sowohl die erhöhte Anzahl Hormon-Signale als auch die Taskumlagerungen wurden in den Logs beobachtet. In diesem Szenario ist so-

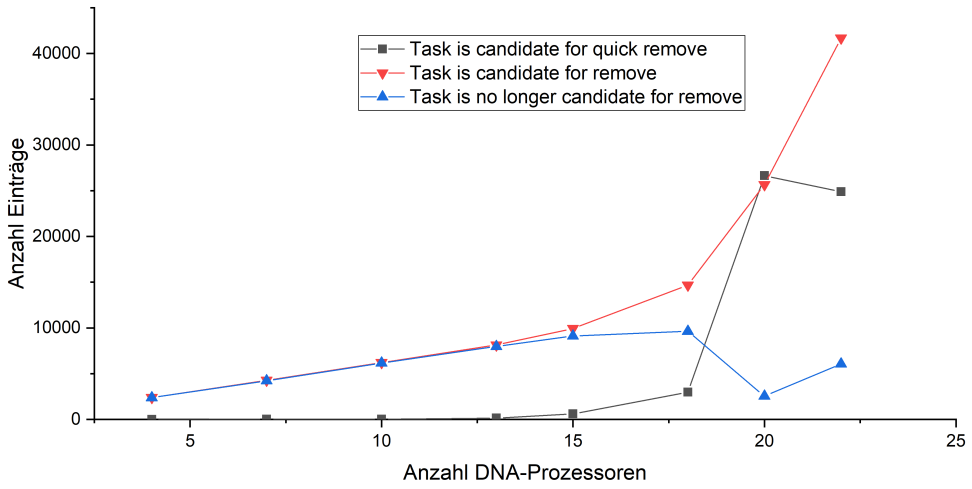

Abbildung 20: Anzahl der Taskumlagerungen für verschiedene Prozessorzahlen

wohl die Überlastung der einzelnen Prozessoren als auch die Überlastung der Kommunikation notwendig, um das irreguläre und schlecht-definierte Verhalten des Systems hervorzurufen. In der Simulationsumgebung mit den virtuellen Prozessoren, bei der alles auf einem einzelnen Gastrechner ausgeführt wird, sind Prozessorüberlastung und Kommunikationsübelastung miteinander verbunden.

5 Abschluss und Ausblick

Das eigentliche Ziel war es, eine autonome Steuerung für Autos aus künstlicher KDNA zu entwickeln und die Möglichkeit zu untersuchen, dass die Autos diese Tasks untereinander verlagern, und das Verhalten des Systems dabei zu untersuchen. Dies sollte einer Situation entsprechen, in der ein Auto mit geringer Leistung einen Teil der Steuerlast auf ein anderes Auto umlagert. Es stellte sich heraus, dass aufgrund der Struktur und Funktionsweise des AHS die Implementierung des kontrollierten Taskaustausches innerhalb der begrenzten Zeit für die Arbeit nicht möglich war. Stattdessen wurde eine intensivere Analyse des Leistungsverhaltens durchgeführt.

Die Kommunikationslast entsteht aufgrund der Task-Verwaltung des AHS.

Schon bei einem Auto auf nur 4 Prozessoren ist die Anzahl an Hormonnachrichten fast genauso hoch wie die Anzahl tatsächlicher Datennachrichten zwischen den Tasks. Die Anzahl Hormonnachrichten skaliert dabei quadratisch in der Anzahl der Prozessorelemente, wobei Eagervalue Hormone die Hauptlast ausmachen, da sie von jeden Prozessor an jeden anderen verschickt werden müssen. Zusätzlich ist die Nachrichtenzahl proportional zur Anzahl Tasks. Die AHS Kommunikation ist also am effizientesten, falls die Anzahl Prozessorelemente möglichst gering und die Auslastung der einzelnen Prozessorelemente damit möglichst hoch ist. Dies stellt ein kritisches Problem für das AHS dar, da die Selbstorganisation auf mehreren heterogen Prozessorelementen, von denen die meisten nicht nahe ihrer Leistungsgrenze sind, das Hauptziel des AHS ist. Es gibt mehrere Möglichkeiten, das Problem anzugehen. Eine Verkleinerung der Anzahl Tasks in der KDNA führt zu einer Reduktion der Last, auch wenn die eigentliche Rechenarbeit sich nicht ändert. Dazu kann man, wie es in der Arbeit gemacht wurde, versuchen Task-intensive Aufgaben aus der KDNA auszulagern oder neue Tasks speziell für diese Aufgaben zu definieren. Das AHS würde dabei vorzugsweise Tasks zu größeren Übertasks automatisch und dynamisch zusammenfassen.

Auch kann die Kommunikation zwischen Prozessoren effizienter gestaltet werden. Die Verarbeitung von hohen Kommunikationslasten in großen Netzwerken ist ein etabliertes Feld. Caching oder eine hierarchische Kommunikationsstruktur könnten das Verhalten vermutlich stark verbessern.

Literatur

[1] Uwe Brinkschulte. An artificial DNA for self-descripting and self-building embedded real-time systems. *Concurrency and Computation: Practice and Experience*, 28(14):3711–3729, 2016.

6 Anhang

```
DNA
File Format Version 1.0

// STEERING CONTROL
100 = 500 (1:103.1 1:106.1) 508 30 // Sensor: Spurabweichung
101 = 500 (1:108.2) 504 30 // Sensor: Strassenrichtung
102 = 70 (1:107.1) 1.570796 30 // Constant: Pi over 2
103 = 1 (1:105.1) $ 0 // ALU: Absolut
105 = 14 (1:106.2) 10 // Offset: + 10
106 = 1 (1:107.2) / 0 // ALU: Division
107 = 1 (1:108.1) * 0 // ALU: Multiplikation
108 = 1 (1:110.1) + 0 // ALU: Addition
109 = 500 (1:110.2) 501 // Sensor: Autorichtung
110 = 1 (1:112.1 1:114.1 1:117.1 1:121.1 1:124.1) - 0 // ALU:
    Subtraktion
112 = 1 (1:113.1) $ 0 // ALU: Absolut
111 = 70 (1:113.2) 3.14159265359 30 // Constant: Pi
113 = 1 (1:114.2 1:115.1) [ 1 // ALU: kleiner gleich
114 = 46 (1:125.1) 0 // Gate
115 = 1 (1:118.2 1:119.2) ! 0 // ALU: NOT
117 = 1 (1:118.1 1:122.1) < 0 // ALU: kleiner als
118 = 1 (1:120.2) & 0 // ALU: AND
122 = 1 (1:119.1) ! 0 // ALU: NOT
119 = 1 (1:123.2) & 0 // ALU: AND
121 = 14 (1:120.1) 6.28319 // Offset: + 2*Pi
120 = 46 (1:125.2) 0 // Gate
124 = 14 (1:123.1) -6.28319 // Offset: - 2*Pi
123 = 46 (1:126.1) 0 // Gate
125 = 1 (1:126.2) + 0 // ALU: Addition
126 = 1 (1:127.1) + 0 // ALU: Addition
127 = 10 (1:128.1 1:132.1) 1.3 0 0.2 30 0 1 // PID
128 = 1 (1:130.1) 2 0 // ALU: Wurzel
130 = 14 (1:131.1) 1 // Offset: + 1
131 = 1 (1:132.2) r 0 // ALU: Wurzel
132 = 1 (1:133.1) / 3 // ALU: Division
```

```
133 = 600 511 // Actor: Steuerrad

// SPEED CONTROL
140 = 500 (1:152.1) 500 30 // Sensor: Geschwindigkeit
141 = 500 (1:143.1) 503 30 // Sensor: Freie Distanz
142 = 500 (1:143.2) 506 30 // Sensor: Maximale Bremskraft
143 = 1 (1:145.1) * 0 // ALU: Multiplikation
144 = 70 (1:145.2) 1.5 30 // Constant: 1.5
145 = 1 (1:147.1 1:148.1) * 0 // ALU: Multiplikation
146 = 70 (1:147.2 1:117.2) 0 30 // Const: 0
147 = 1 (1:148.2) ] 0 // ALU: groesser gleich
148 = 46 (1:149.1) 0 // GATE
149 = 1 (1:151.2) r 0 // ALU: Wurzel
150 = 500 (1:151.1) 507 30 // Sensor:
    Geschwindigkeitsbegrenzung
151 = 1 (1:152.2) m 0 // ALU: Minimum
152 = 1 (1:153.1 1:154.1) > 0 // ALU: groesser als
153 = 600 512 // Actor: Bremsen
154 = 1 (1:155.1) ! 0 // ALU: NOT
155 = 600 510 // Actor: Gaspedal

929 = 998 (1:930.1) 100 // DNA Checker
930 = 600 599 // Actor: Notstopp
```