UNIKASSEL
VERSITÄT

Vergleich zweier experimenteller Laufzeitsysteme anhand der Auswertung von GNNs

Bachelorarbeit

Vorgelegt im Fachbereich 16 – Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik
der Universität Kassel

von Luca Hertel
35380724

Erstgutachterin:
Prof. Dr. Claudia Fhory

Zweitgutachter:
Dr. Josephine Thomas

Eingereicht am 18. August 2023 in Kassel

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Kassel, 18. August 2023

Luca Hertel

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren hat sich die Kernanzahl von CPUs immer mehr erhöht. So ist es heute keine Besonderheit, dass normale Verbraucher zu Hause einen Rechner mit acht oder mehr CPU-Kernen besitzen [1].
Durch die erhöhte Anzahl von Kernen wird die Leistung der Rechner direkt verbessert, wenn mehrere Programme gleichzeitig ausgeführt werden. Allerdings ist diese Leistungsverbesserung bei der Ausführung mehrerer Programme nicht direkt auf die Leistung bei der Ausführung eines einzigen Programms übertragbar.
Um auch eine Leistungssteigerung bei einzelnen, sehr aufwändigen Programmen erzielen zu können, kann taskbasierte Parallelverarbeitung verwendet werden. Dabei werden Programme in viele kleine Teilaufgaben zerlegt und auf mehrere CPU-Kerne verteilt. Dafür ist jedoch zusätzlicher Aufwand bei der Entwicklung und Ausführung der Programme notwendig.
Bei der Entwicklung muss entschieden werden, mit welcher Bibliothek diese Programme parallelisiert werden und wie die Programme in Teilaufgaben zerlegt werden. Außerdem ist für die Aufteilung der Programme zusätzliche Laufzeit nötig.
Futures sind ein Programmierkonzept, welche die Parallelisierung von komplexen Berechnungen mit vielen Abhängigkeiten ermöglichen.
Zwei Programmiersysteme, welche die Verwendung von Futures unterstützen, sind Cilk-F und Spidroin.
Spidroin ist ein neues paralleles Programmiersystem, welches zum Zeitpunkt dieser Arbeit an der Universität Kassel entwickelt wird. Cilk-F ist eine angepasste Version des parallelen Programmiersystems Cilk, welche um Futures erweitert wurde [2, 3].
Bei der Entwicklung von Spidroin ist der Vergleich mit einem ähnlichen Programmiersystem interessant.

Für den Vergleich wird ein rechenintensives Programm benötigt. Graph neuronale Netze (GNN) sind Programme, welche Daten in Form von Graphen als Eingabe entgegennehmen und in der Lage sind, diese Daten zu interpretieren und danach Aussagen über den Graphen zu treffen. GNNs sind ein sehr aktuelles Forschungsthema und werden im Bereich des maschinellen Lernens angewendet.
In dieser Arbeit werden die beiden experimentellen futurebasierten parallelen Programmiersysteme Cilk-F und Spidroin anhand der Auswertung von GNNs verglichen.
Die Fragestellung ist, wie verhalten sich die Laufzeit und der Speedup von Spidroin im Vergleich zu Cilk-F? Außerdem wird die Benutzbarkeit diskutiert.
Der Speedup ist der Faktor, um den sich die Laufzeit eines Programms verringert, wenn mehr Kerne verwendet werden [4].
Dazu wird die Ausführungszeit des Programms gemessen. Diese verhält sich im Optimalfall antiproportional zu der Anzahl von verwendeten CPU-Kernen. Um dieses Verhalten zu analysieren, werden Programme verwendet, welche als Benchmarks bezeichnet werden.
Für den Vergleich der beiden Programmiersysteme wurde ein Benchmark mit beiden Systemen implementiert, welcher ein GNN auswertet. Danach wurden die Ausführungszeiten der beiden Benchmarkversionen mit verschiedenen Parametern und verschiedenen Anzahlen an verwendeten CPU-Kernen gemessen. Dabei hat sich herausgestellt, dass Cilk-F geringere Laufzeiten und einen besseren Speedup bietet. Cilk-F hat in jedem Punkt bei jedem Test besser abgeschnitten, wobei der Unterschied der beiden Programmiersysteme stark von den Benchmarkparametern abhängig ist.

In Kapitel 2 werden die Grundlagen zu Parallelverarbeitung (2.1), Futures (2.2), Spidroin (2.3), Cilk-F (2.4) und GNNs (2.5) erklärt. Daraufhin beschreiben wir in Kapitel 3 den Benchmark. Anschließend werden die Durchführung und Ergebnisse des Performancevergleichs in Kapitel 4 dargestellt und diskutiert und zum Schluss folgt in Kapitel 5 ein Fazit.

2 Grundlagen

In diesem Kapitel stellen wir die notwendigen Grundlagen der Arbeit vor. Abschnitt 2.1 geht auf taskbasierten Parallelverarbeitung ein. Anschließend wird in Abschnitt 2.2 das Konzept von Futures beschrieben. In den Abschnitten 2.3 und 2.4 werden die Grundlagen von Spidroin und Cilk-F erklärt und abschließend werden in Kapitel 2.5 GNN vorgestellt.

2.1 Taskbasierte Parallelverarbeitung

Bei der Parallelverarbeitung wird ein Programm in Teilaufgaben zerlegt. Dadurch ist es möglich, die verschiedenen Teilaufgaben auf mehreren CPU-Kernen parallel zu berechnen, sofern diese Teilaufgaben unabhängig voneinander sind. Dazu wird oft manuell ein eigener Thread für eine Teilaufgabe erzeugt.
Das Besondere an taskbasierter Parallelverarbeitung ist, dass ein Programm in *viele* kleinere Teilaufgaben, welche *Tasks* genannt werden, zerlegt wird [5]. Dadurch, dass viele Tasks erzeugt werden, welche selber keine aufwändigen Berechnungen darstellen, ist es nicht effizient, für jeden Task einen eigenen Thread zu erzeugen. Daher werden bei der taskbasierten Parallelverarbeitung *Worker* eingesetzt. Worker sind dabei Threads, welche bei der Initialisierung des Programmiersystems erzeugt werden und mehrere Tasks bearbeiten. Worker besitzen dafür oft eine eigene Warteschlange mit Tasks und das Programmiersystem verteilt Tasks automatisch auf die Worker. Außerdem gibt es meistens auch eine Möglichkeit, dass Tasks zwischen den Warteschlangen der Worker verschoben werden können, damit alle Worker effizient genutzt werden. Eine Möglichkeit dafür ist, dass Worker sich gegenseitig Tasks stehlen.
Bei der taskbasierten Parallelverarbeitung wird zusätzlicher Aufwand für das gleich-

mäßige Verteilen der Tasks auf die Worker betrieben, was auch Load Balancing genannt wird. Die Implementierung des Load Balancings ist ein entscheidender Faktor für die Performance eines Programmiersystems. Die Implementierung beeinflusst, wie viele Tasks jeder Worker zur Bearbeitung hat und den Aufwand, der für die Verteilung betrieben werden muss. Dieser Aufwand kann mit einer höheren Anzahl an Workern ansteigen, was sich auf den Speedup, die Effizienz und die Skalierbarkeit auswirkt.

2.2 Futures

Futures sind ein Programmierkonzept, welches häufig für Aufgaben verwendet wird, die parallel ausgeführt werden sollen. Futures repräsentieren einen Wert, der nicht von Anfang an verfügbar ist, aber noch bereitgestellt wird. Ein Future kann zwei Zustände haben: Entweder ist die Berechnung noch nicht abgeschlossen, was bedeutet, dass der Wert noch nicht zur Verfügung steht oder sie ist abgeschlossen und der Wert kann ausgelesen werden. Zur Verwendung von Futures werden von Programmiersystemen häufig folgende oder ähnliche Funktionen bereitgestellt:

- `spawn`: Die `spawn`-Funktion wird verwendet, um ein Future zu erzeugen. Als Argumente bekommt sie in der Regel eine Funktion, die ausgeführt werden soll, sowie die dafür vorgesehenen Parameter.
- `wait`: Die `wait`-Funktion kann auf Futures aufgerufen werden. Ein Aufruf führt dazu, dass die Ausführung des aufrufenden Tasks pausiert wird, bis der Wert verfügbar ist.
- `get`: Die `get`-Funktion wird verwendet, um den Rückgabewert des Futures zu erhalten. Je nach Programmiersystem muss vorher `wait` verwendet werden.

2.3 Spidroin

Spidroin ist ein paralleles Programmiersystem, welches zum Zeitpunkt dieser Arbeit am Fachgebiet Programmiersprachen/-methodik des Fachbereichs 16 der Universität

Kassel entwickelt wird. Spidroin stellt Futures zur Verfügung, welche auf C++ *Coroutines* basieren. Coroutines sind Funktionen, welche unterbrochen und später fortgesetzt werden können [6]. Für die Abarbeitung der Futures werden Worker verwendet, welche sich gegenseitig Arbeit stehlen.
Um die Beispielfunktion `add` aus Listing 2.4 mithilfe eines Spidroin Futures zu berechnen, muss die Funktion angepasst werden.

```
1 spid::task_coro<int> addCoro(int a, int b) {
2     co_return a + b;
3 }
```

Listing 2.1: Beispielfunktion welche mit Spidroin Futures ausgeführt werden soll

Listing 2.1 zeigt die angepasste Version `addCoro` von `add`. Funktionen, welche in Tasks mit Spidroin ausgeführt werden sollen, müssen immer den Rückgabetyp `spid::task_coro<Type>` besitzen. Dabei ist `Type` der tatsächliche Rückgabetyp der Funktion. Anstelle eines `return`-Statements muss ein `co_return` verwendet werden.

```
1 spid::init( numberOfWorkers );
2 std::shared_ptr<spid::future<int>> future = spid::spawn(addCoro, 1,
      2);
3 future->wait();
4 spid::tini();
5 int result = future->get();
```

Listing 2.2: Verwendung von Spidroin Futures in normalen Funktionen

Listing 2.2 zeigt die Verwendung eines Spidroin Futures. In Zeile 1 erfolgt die Initialisierung des Laufzeitsystems, wobei die Anzahl von Workern mit dem Argument `numberOfWorkers` angegeben wird. In Zeile 2 wird ein `spid::future` mit der Funktion `addCoro` und ihren Parametern 1 und 2 erzeugt und an der Adresse `future` gespeichert.
Danach wird in Zeile 3 die `wait`-Funktion auf dem Future aufgerufen. Dies führt dazu, dass gewartet wird, bis das Future abgeschlossen ist. Allerdings ist `wait` eine blockierende Funktion, was bedeutet, dass der aufrufende Thread nichts macht, bis das Future abgeschlossen ist. Eine nicht blockierende Variante wird in Listing 2.3 gezeigt.
Anschließend wird in Zeile 4 das Laufzeitsystem abgebaut und in Zeile 5 der Wert

des Futures abgerufen. Wichtig ist, dass `get` erst auf ein Future angewendet werden darf, nachdem auf das Future gewartet wurde. Das bedeutet, im Gegensatz zu der `get`-Funktion von Cilk-F erfüllt die Spidroinvariante nur den in Abschnitt 2.2 definierten `get`-Teil.
Für den Fall, dass mehrere Futures gleichzeitig berechnet werden sollen und dabei Werte von anderen Futures benötigt werden, ist diese Variante keine Option. Stattdessen sollte eine Hilfsfunktion verwendet werden.

```
1 spid::task_coro<void> helpCoro() {
2     std::shared_ptr<spid::future<int>> future = spid::spawn(
    addCoro, 3, 4);
3     co_await future;
4     int result = future->get();
5 }
```

Listing 2.3: Verwendung von Spidroin Futures in **co_routines**

Listing 2.3 zeigt, wie auf ein Future gewartet werden kann, ohne dabei den Thread zu blockieren. Dazu muss die Hilfsfunktion `helpCoro` implementiert werden, welche selber eine Coroutine ist.
In Zeile 2 wird wie in Listing 2.2 ein Future erzeugt. Der Unterschied ist, dass in dieser Variante `co_await` (Zeile 3) verwendet wird, um auf das Future zu warten. Das Keyword `co_await` kann nur in Coroutinen verwendet werden, weshalb die Hilfsfunktion nötig ist. Der Vorteil ist, dass der Task an der Stelle, wo `co_await` aufgerufen wird, pausiert wird, anstatt, dass der ganze Thread pausiert wird.
Danach kann normal auf den Wert des Futures mit `get` zugegriffen werden. Die Hilfsfunktion `helpCoro` kann nach dem Schema von Listing 2.2 an einer beliebigen Stelle in einem Future ausgeführt werden.

2.4 Cilk-F

Cilk-F ist ein paralleles Programmiersystem, welches die Verwendung von Futures unterstützt und 2019 entwickelt wurde. Es ist jedoch ein Prototyp und weist daher bei der Programmierung ein paar Besonderheiten auf. Cilk-F verwendet Worker für die Ausführung von Tasks. Die Worker stehlen automatisch sofort Tasks von anderen

Workern, sobald der aktuell ausgeführte Task pausiert wird. Dies ist der Fall, wenn beispielsweise auf ein Future gewartet werden muss. Dieses Vorgehen wird Proactive Workstealing genannt.
Aufgrund des Entwicklungsstandes von Cilk-F muss bei der Verwendung ein zusätzlicher Aufwand betrieben werden.

```
1 int add(int a, int b)
2 {
3     return a + b;
4 }
```

Listing 2.4: Beispielfunktion welche mit Cilk-F Futures ausgeführt werden soll

Listing 2.4 zeigt eine Funktion namens `add`, welche zwei `int` entgegennimmt und die Summe der beiden Parameter als `int` zurückgibt. Wenn diese Funktion parallel mithilfe von Futures in Cilk-F ausgeführt werden soll, muss dafür eine Hilfsfunktion von dem Anwender implementiert werden.

```
1 void __attribute__((noinline)) add_fut_helper(cilk::future<int> *
    fut, int a, int b) {
2     FUTURE_HELPER_PREAMBLE;
3     void *__cilkrts_deque = fut->put(add(a, b));
4     if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque
    , 2);
5     FUTURE_HELPER_EPILOGUE;
6 }
```

Listing 2.5: Cilk-F Hilfsfunktion für die Ausführung mit Futures

Listing 2.5 zeigt die Hilfsfunktion `add_fut_helper`, welche benötigt wird, um `add` in einem Task auszuführen. Hilfsfunktionen haben immer den Rückgabetyp `void`. Des Weiteren müssen sie immer das Attribut `noinline` besitzen. Außerdem besitzt die Funktion `add_fut_helper` drei Parameter. Der erste Parameter `fut` ist ein Zeiger auf ein Cilk-Future, welches einen `int` enthält. Bei den anderen beiden Parametern handelt es sich um die beiden `int`, welche an `add` übergeben werden.
In den Zeilen 2 und 5 werden Makros verwendet, welche von Cilk-F bereitgestellt werden. Das Makro `FUTURE_HELPER_PREAMBLE` muss am Anfang jeder Hilfsfunktion verwendet werden und übernimmt einige Initialisierungsaufgaben. Das Makro

`FUTURE_HELPER_EPILOGUE` muss am Ende jeder Hilfsfunktion verwendet werden und schließt die Initialisierung ab.
In Zeile 3 folgt nun der Funktionsaufruf von `add`. Falls `add` den Rückgabetyp `void` hätte, würde der Funktionsaufruf nicht in `put` passieren, sondern könnte als normaler Funktionsaufruf zwischen den Zeilen 2 und 3 eingefügt werden. Bei der Verwendung von `void`-Futures kann trotzdem die `wait`-Funktion ausgenutzt werden.

```
1  int main()
2  {
3      CILK_FUNC_PREAMBLE;
4
5      cilk::future<int> * fut_1;
6      cilk::future<int> * fut_2;
7
8      START_FIRST_FUTURE_SPAWN;
9      add_fut_helper(&fut_1, 1, 2);
10     END_FUTURE_SPAWN;
11
12     START_FUTURE_SPAWN;
13     add_fut_helper(&fut_2, 3, 4);
14     END_FUTURE_SPAWN;
15
16     int value_1 = fut_1.get();
17     int value_2 = fut_2.get();
18
19     CILK_FUNC_EPILOGUE;
20
21     return 0;
22     }
```

Listing 2.6: Cilk-F Futureerstellung und Verwendung

In Listing 2.6 wird gezeigt, wie die Funktion `add` zweimal mithilfe von Cilk-F Futures ausgeführt wird. Dazu wird in Zeile 3 das Makro `CILK_FUNC_PREAMBLE` verwendet, welches Initialisierungsaufgaben für das Laufzeitsystem übernimmt. Dieses Makro muss verwendet werden, bevor andere Cilk-F Funktionalitäten verwendet werden können.

In den Zeilen 4 und 5 werden zwei `cilk::future<int> *` namens `fut_1` und `fut_2` deklariert. Beide enthalten dabei einen `int`, da dieser Datentyp von `add` zurückgegeben wird.
Danach werden die beiden Futures erzeugt. Dazu wird in Zeile 9 die Hilfsfunktion für `add` mit den Argumenten `fut_1`, 1 und 2 aufgerufen. Dadurch wird ein Future an der Adresse `fut_1` initialisiert, welches die Funktion `add(1,2)` berechnet. Jeder Aufruf einer Hilfsfunktion muss dazu von den Makros `START_FUTURE_SPAWN` und `END_FUTURE_SPAWN` umschlossen werden. Handelt es sich dabei um den ersten Aufruf einer Hilfsfunktion im Laufe des Programms, so muss das Makro `START_FIRST_FUTURE_SPAWN` anstelle von `START_FUTURE_SPAWN` verwendet werden, was in den Zeilen 8 und 12 zu sehen ist. Der Aufruf einer Hilfsfunktion und die beiden Makros, welche diese umschließen, wird als Spawnen eines Futures bezeichnet. Nach diesen drei Zeilen wird die Berechnung der hinterlegten Funktion gestartet.
Nachdem ein Future gespawnt wurde, kann die Funktion `get` auf dieses Future angewendet werden. Diese Funktion vereint dabei die beiden typischen Futurefunktionen `wait` und `get`, welche in Abschnitt 2.2 erwähnt wurden. In Zeile 16 wird `fut_1.get()` aufgerufen und der Rückgabewert in einem `int` namens `value_1` gespeichert. Das führt dazu, dass der Task der Funktion `main` pausiert wird, bis das Future abgeschlossen ist. Der Wert, welcher `value_1` zugewiesen wird, entspricht dem Ergebnis der Addition von 1 und 2 (Zeile 9).
Abschließend wird das Makro `CILK_FUNC_EPILOGUE` verwendet, um das Laufzeitsystem zu beenden. Bei der Arbeit mit Cilk-F ist es sehr wichtig, die Makros an den richtigen Stellen zu verwenden. Wird ein Makro vergessen oder zu oft verwendet, kommt es zu Speicherzugriffsfehlern.

2.5 GNNs

Neuronale Netze bestehen aus einer Vielzahl an sogenannten Neuronen, welche auf mehrere Ebenen aufgeteilt sind. Die Funktion eines Neurons ist es, mehrere Parameter von Neuronen aus der vorherigen Ebene zu bekommen und daraus eine Ausgabe zu berechnen. So sind neuronale Netze in der Lage, eine große Menge von

Parametern zu interpretieren und auf deren Grundlage Aussagen zu treffen. Um die Genauigkeit der Aussagen zu verbessern, müssen neuronale Netze trainiert werden, wozu in der Regel große Datensätze benötigt werden. Dabei werden die Aussagen des neuronalen Netzes mit der Realität verglichen und die Berechnungen der Neuronen angepasst [7].
Eine besondere Form der neuronalen Netze sind Graph neuronale Netze [8]. Diese unterscheiden sich in ihrer Struktur von den klassischen neuronalen Netzen, da es keine Neuronen gibt. Stattdessen basieren GNNs auf Graphen. Das bedeutet, es gibt mehrere Knoten $v \in V$, welche mit Kanten verbunden sein können. In dieser Arbeit beschränken wir uns auf ungerichtete Graphen, was bedeutet, dass Kanten zwischen zwei Knoten immer in beide Richtungen gehen. Die Funktion von Knoten ähnelt der von Neuronen. Sie bekommen ebenfalls mehrere Parameter und verwenden diese in ihrer Berechnung. Allerdings hat jeder Knoten eine Reihe an Informationen, welche diesen beschreiben. Diese Informationen werden Feature-Vektor X_V genannt.
Die Eingabe und Ausgabe eines GNN sind jeweils ein Graph. Diese beiden Graphen unterscheiden sich nur durch die Feature-Vektoren. Es werden weder Kanten noch Knoten hinzugefügt oder entfernt. Die Berechnung eines GNN ist in mehrere Phasen unterteilt. Allerdings gibt es keine Ebenen zwischen denen Informationen weitergereicht werden, sondern mehrere Zeitschritte, welche verschiedene Versionen des Graphen darstellen.
Die Berechnung des nächsten Zeitschrittes erfolgt, indem jeder Knoten Informationen über seine Nachbarn sammelt und daraus einen neuen eigenen Feature-Vektor berechnet. Dieser Vorgang wird bei jedem Zeitschritt für jeden Knoten wiederholt. Die vollständige Berechnung aller Zeitschritte und somit der finalen Version des Graphen wird im Folgenden als Auswertung eines GNN bezeichnet. Während eines Zeitschritts lernen die Knoten etwas über die Eigenschaften ihrer Nachbarn und über mehrere Zeitschritte hinweg auch indirekt etwas über Knoten, welche keine direkten Nachbarn sind. Es gibt auch Versionen von GNNs, welche den Kanten oder dem gesamten Graphen Feature-Vektoren zuordnen, welche in dieser Arbeit nicht verwendet werden. Da es in dieser Arbeit nicht darum geht, ein GNN zu trainieren und präzise Aussagen zu erhalten, sondern um den Vergleich zweier paralleler Programmiersysteme, verwenden wir im Folgenden nur GNNs welche nur

Feature-Vektoren für Knoten besitzen.

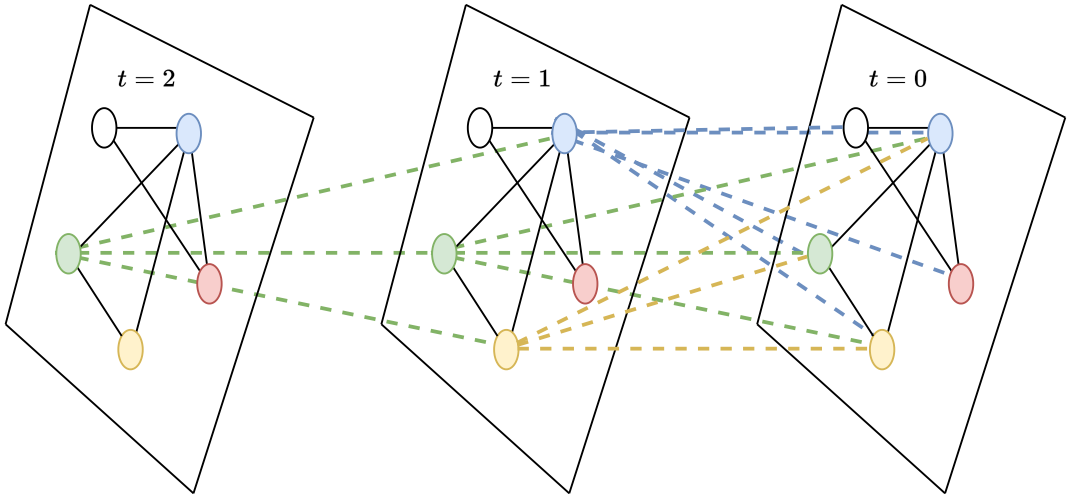

Abbildung 2.1: GNN Informationsfluss

Abbildung 2.1 zeigt, wie sich Informationen von Knoten über mehrere Zeitschritte hinweg verteilen. Die drei Ebenen stellen dabei den Graphen zu verschiedenen Zeitschritten dar, wobei rechts der Zeitschritt t_0 ist. Die Linien innerhalb einer Ebene sind Kanten zwischen den Knoten. Diese bleiben bei jedem Zeitschritt gleich. Die gestrichelten Linien zwischen den Ebenen zeigen an, von welchem Knoten Informationen verwendet werden.
In der Abbildung wird der Informationsfluss für den grünen Knoten dargestellt. Dieser hat zwei direkte Nachbarn. Das bedeutet, dass er für die Berechnung seines neuen Feature-Vektors den eigenen Vektor und die Vektoren der beiden Nachbarn verwendet. Bei Betrachtung der linken Ebene (Zeitschritt t_2) ist zu sehen, dass für die Berechnung von Zeitschritt t_1 zu t_2 nur Informationen von drei Knoten verwendet werden. Wenn jedoch die Berechnung von Zeitschritt t_0 zu t_2 betrachtet wird, ist zu sehen, dass dafür Informationen von allen Knoten verwendet werden. Auf diese Weise erhalten Knoten bei jedem Zeitschritt Informationen über weiter entfernte Knoten. Für die Berechnung des nächsten Zeitschritts muss für jeden Knoten der neue Feature-Vektor berechnet werden.

Das genaue Vorgehen für die Berechnung eines neuen Feature-Vektors X_v^t kann sehr unterschiedlich sein und wird in dieser Arbeit wie folgt gewählt:

- X_v^t: Der Feature-Vektor des Knotens v zum Zeitpunkt t
- σ: Die Updatefunktion welche den Finalen Wert des neuen Feature-Vektors berechnet
- W_t: Eine Matrix zur Gewichtung der aggregierten Nachbarinformationen
- $a(v,t)$: Die Aggregierungsfunktion zum Zusammenfassen der Nachbarinformationen
- B_t: Eine Matrix zur Gewichtung der eigenen Informationen

$$X_v^t = \sigma\left(W_t * a(v,t) + B_t * X_v^{t-1}\right) \tag{2.1}$$

Als Updatefunktion wurde in dieser Arbeit die Sigmoid-Funktion gewählt:

$$\sigma(x) = \frac{1}{q + e^{-x}}$$

Die Updatefunktion wird dabei auf jedes Element eines Feature-Vektors angewendet und sorgt dafür, dass jeder Wert eines Feature-Vektors zwischen 0 und 1 liegt.
Das Zusammenfassen der Nachbarvektoren in der Aggregierungsfunktion besteht daraus, den Durchschnitt der Nachbarvektoren zu berechnen:

$$a(v,t) = \sum_{u \in N(v)} \frac{X_u^{t-1}}{|N(v)|} \tag{2.2}$$

Dabei beschreibt $N(v)$ die Menge der Nachbarknoten von v.

3 Beschreibung des Benchmarks und Implementierungen

In diesem Kapitel wird auf den Benchmark und seine beiden Versionen eingegangen. Abschnitt 3.1 beschreibt den Ansatz zur parallelen Auswertung eines GNN. In Abschnitt 3.2 folgt die Implementierung mit Spidroin und danach die Implementierung mit Cilk-F in Abschnitt 3.3. Abschließend wird in Abschnitt 3.4 analysiert, wie sich die Parameter theoretisch auf die Laufzeit auswirken, um dies für die Interpretation der Ergebnisse in Kapitel 4 zu nutzen.
Als Benchmark wird die Auswertung eines GNNs verwendet.

3.1 Parallelisierungsansatz

Die Eingabe eines GNNs ist ein Graph mit initialen Feature-Vektoren für seine Knoten. Dieser Graph stellt den Zeitpunkt t_0 dar. Zur Berechnung von t_n werden laut der Aggregierungsfunktion (2.2) Feature-Vektoren aus t_{n-1} verwendet. Mit der Berechnung von t_1 kann direkt gestartet werden. Für die Berechnung von t_n mit $n > 1$ müssen vorher die Feature-Vektoren aus t_{n-1} zur Verfügung stehen.
Es gibt viele Möglichkeiten, die Auswertung eines GNNs zu parallelisieren. In dieser Arbeit wird die Auswertung eines GNNs realisiert, indem jede Berechnung eines Feature-Vektors X_v^t in einem eigenen Task ausgeführt wird. Das bedeutet, es wird für jede dieser Berechnungen ein Future erzeugt.
Es wird zugelassen, dass Feature-Vektoren aus verschiedenen Zeitschritten parallel berechnet werden. Dabei kann es dazu kommen, dass für die Berechnung eines Feature-Vektors Werte benötigt werden, welche noch nicht zur Verfügung stehen.

Dafür bietet sich die Verwendung von Futures an, da die Abhängigkeiten der einzelnen Tasks durch das Warten auf das zugehörige Future gelöst werden können. Zu Beginn des Benchmarks werden alle Futures für jeden Knoten zu jedem Zeitschritt erzeugt. Die genaue Ausführungsreihenfolge der Futures bleibt damit dem Laufzeitsystem überlassen.

3.2 Spidroin-Implementierung

Listing 3.1 zeigt die Funktion `calculate`, welche die Futures spawnt und wartet, bis diese abgeschlossen sind.
In Zeile 3 wird ein `spid::promise<bool>` namens `initDone` angelegt. Ein *promise* repräsentiert den Wert sowie die Vollendung oder den Fehlschlag einer Operation, welche parallel ausgeführt wird. Es wird später in `calculateNextNodeStateCoro` verwendet, um die Berechnungen der Knotenzustände nach der Initialisierung aller Futures zu starten.
In den Zeilen 4 bis 10 werden die Futures für die Berechnungen der Knotenzustände erzeugt und in `futureArray` gespeichert. Dabei wird der Funktion zur Berechnung der Knotenzustände als letztes Argument das Future von `initDone` übergeben.
Nachdem alle Futures erzeugt wurden, wird in Zeile 11 der Wert von `initDone` auf `true` gesetzt, wodurch das zugehörige Future als abgeschlossen angesehen wird.
Abschließend wird in den Zeilen 12 bis 18 gewartet, bis alle Futures abgeschlossen sind.

```
1 spid::task_coro<bool> calculate (int numOfWorkers) {
2 {
3     spid::promise<bool> initDone;
4     for(uint node = 0; node < NUM_NODES; node++)
5     {
6         for(uint time = 0; time < NUM_CALCULATIONS; time++)
7         {
8             futureArray[time][node] = std::move(spid::spawn(
    calculateNextNodeStateCoro, time, node, initDone.get_future()));
9         }
```

```
10      }
11      initDone.set(true);
12      for(uint time = 1; time < NUM_CALCULATIONS; time++)
13      {
14          for(uint node = 0; node < NUM_NODES; node++)
15          {
16                  co_await futureArray[time][node];
17          }
18      }
19      co_return true;
20  }
```

Listing 3.1: Erstellung der Futures

Die Funktion `calculateNextNodeStateCoro` wurde wie folgt implementiert:
In Zeitschritt $t = 0$ werden alle Werte der Featurevektoren auf 1 gesetzt.
Wenn es sich nicht um den initialen Zustand handelt, muss der Zustand mit Hilfe von Zuständen aus dem vorherigen Zeitschritt berechnet werden.

```
1  co_await initDone;
2  if(times > 0)
3  {
4      co_await futureArray[times -1][node];
5  }
6  std::array<double, NUM_FEATURES> neighbourFeatures;
7  std::array<double, NUM_FEATURES> ownFeatures;
8  int neighbourCount = 0;
9  for (int neighbour = 0; neighbour < NUM_NODES; neighbour++)
10 {
11     if(adjacencyMatrix[node][neighbour] == true)
12     {
13         neighbourCount++;
14         co_await futureArray[times-1][neighbour];
15         for(int feature = 0; feature < NUM_FEATURES; feature++)
16         {
17             neighbourFeatures[feature] += futureArray[times-1][
    neighbour]->get()[feature];
18         }
19     }
```

```
20 }
21 for(int feature = 0; feature < NUM_FEATURES; feature++)
22 {
23     neighbourFeatures[feature] = neighbourFeatures[feature] /
       neighbourCount;
24 }
25 multiply(weights, neighbourFeatures);
```

Listing 3.2: Nachbarfeatures aggregieren

In Listing 3.2 wird gezeigt, wie die Featurevektoren der Nachbarknoten verarbeitet werden. Da sichergestellt werden muss, dass alle Futures bereits initialisiert wurden, wird in Zeile 1 auf das Future `initDone` gewartet, welches bereits in 3.1 (Zeile 8) gezeigt wurde. Danach wird in Zeile 4 auf den eigenen Vorgänger gewartet.
In den Zeilen 9 bis 20 wird über alle Knoten iteriert. Falls es sich um einen Nachbarn des aktuellen Knotens handelt (Zeile 11), wird gewartet, bis das Future des Nachbarn aus dem vorherigen Zeitschritt abgeschlossen ist. Danach werden die Featurevektoren der Nachbarn zusammenaddiert (Zeile 17).
In den Zeilen 21 bis 24 werden die addierten Featurevektoren durch die Anzahl an Nachbarn geteilt, wodurch der Durchschnitt aller Featurevektoren der Nachbarknoten berechnet wird. Dieser Durchschnittsvektor wird abschließend in Zeile 25 mit einer Gewichtungsmatrix multipliziert.
Zur Berechnung des neuen Featurevektores wird zudem noch der alte eigene Zustand verwendet.

```
1  for(int feature = 0; feature < NUM_FEATURES; feature++)
2  {
3      ownFeatures[feature] += futureArray[times-1][node]->get()[
       feature];
4  }
5  multiply(bias, ownFeatures);
6  Features array;
7  for(int feature = 0; feature < NUM_FEATURES; feature++)
8  {
9      double a = 1.0/(1.0 + exp(-(neighbourFeatures[feature] +
       ownFeatures[feature])));
10     array[feature] = a;
```

```
11 }
12 co_return array;
```

Listing 3.3: Berechnung des neuen Featurevektors

Der letzte Teil der Berechnung des neuen Featurevektores wird in Listing 3.3 gezeigt. In den Zeilen 1 bis 5 wird der alte eigene Featurevektor kopiert und mit einer Gewichtungsmatrix multipliziert. Dieser Vektor wird mit den zusammengefassten Nachbarvektoren addiert. Anschließend wird auf jedes Element des Vektors die Sigmoidfunktion angewendet, was in den Zeilen 7 bis 11 zu sehen ist. Der neu berechnete Vektor wird in Zeile 12 zurückgegeben und stellt den Wert des Futures dar.

3.3 Cilk-F-Implementierung

Die Implementierung mit Cilk-F ist größtenteils identisch mit der in Kapitel 3.2 beschriebenen Spidroin-Variante. Daher wird in diesem Abschnitt nur auf die Unterschiede eingegangen, wobei die syntaktischen Unterschiede weggelassen werden, da diese bereits in Kapitel 2 beschrieben wurden.

```
1  void calculate()
2  {
3    const int numFutures = NUM_CALCULATIONS*NUM_NODES;
4    CILK_FUNC_PREAMBLE;
5    Future futures = new cilk::future<void>[numFutures];
6    START_FIRST_FUTURE_SPAWN;
7    fut_helper(&futures[0], 0, 0, futures);
8    END_FUTURE_SPAWN;
9    for (uint time = 0; time < NUM_CALCULATIONS; time++)
10   {
11     for(uint node = 0; node < NUM_NODES; node++)
12     {
13       if((time != 0) || (node != 0))
14       {
15         START_FUTURE_SPAWN;
```

```
16          fut_helper(&futures[(time*NUM_NODES) + node], time, node,
    futures);
17          END_FUTURE_SPAWN;
18        }
19      }
20    }
21    for (uint time = 0; time < NUM_CALCULATIONS; time++)
22    {
23      for(uint node = 0; node < NUM_NODES; node++)
24      {
25        futures[(time*NUM_NODES) + node].get();
26      }
27    }
28      CILK_FUNC_EPILOGUE;
29  }
```

Listing 3.4: Erstellung der Futures mit Cilk-F

Listing 3.4 zeigt die Funktion `calculate`, welche auch in der Cilk-F-Variante zur Erzeugung der Futures verwendet wird.
Ein Unterschied ist in Zeile 5 zusehen. Das dort erstellte Array `futures` wird verwendet, um die Futures zu verwalten. Dieses Array war in der Spidroin-Variante global, was bei Cilk-F zu Problemen geführt hat. Daher wird das Array lokal erstellt und der Pointer weitergereicht.
Außerdem sind die Futures vom Typ `cilk::future<void>` anstatt von `cilk::future<std::array<double, NUM_FEATRURES>>`. Es ist möglich in Cilk-F Futures mit einem Wert zu verwenden, allerdings wird nicht jeder Datentyp unterstützt. Primitive Datentypen, wie `int` können verwendet werden, jedoch ist die Verwendeung von `std::array` nicht möglich. Daher wird für die Speicherung der Featurevektoren ein eigenes Array verwendet.
In den Zeilen 6 bis 20 werden die Futures im Stil von Cilk-F erzeugt, wobei die Hilfsfunktion `fut_helper` verwendet wird und in den Zeilen 21 bis 28 wird gewartet, bis alle Futures abgeschlossen sind.

```
1  void __attribute__((noinline)) fut_helper(Future fut, uint time,
     uint node, Future futures) {
2     FUTURE_HELPER_PREAMBLE;
```

```
3     calculateNextNodeState(time, node, futures);
4     void *__cilkrts_deque = fut->put();
5     if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque
      , 2);
6     FUTURE_HELPER_EPILOGUE;
7 }
```

Listing 3.5: Cilk-F Hilfsfunktion

Die Hilfsfunktion `fut_helper` wird in Listing 3.5 gezeigt. Da die Funktion `calculateNextNodeState` in diesem Fall keinen Rückgabewert hat und die Futures daher vom Typ `void` sind, wird die Funktion in Zeile 3 normal aufgerufen, anstatt in Zeile 4 `put()` übergeben zu werden. Abgesehen von dem fehlenden Rückgabewert unterscheidet sich `calculateNextNodeState` in der Cilk-F-Variante nicht von der Spidroin-Variante.

3.4 Theoretischer Einfluss der Benchmarkparameter

In diesem Abschnitt wird hergeleitet, wie die Anzahl der Zeitschritte (`NUM_CALCULATIONS`), die Anzahl der Knoten (`NUM_NODES`) und die Anzahl der Features (`NUM_FEATURES`) die Laufzeit des Benchmarks beeinflussen sollten. Die Abschätzungen der Laufzeiterhöhungen werden in Kapitel 4 mit den tatsächlichen Laufzeiterhöhungen verglichen.
Die Anzahl an Zeitschritten sollte sich proportional zur Laufzeit verhalten. Werden die Zeitschritte erhöht, wird die Anzahl der Ausführungen von `calculateNextNodeState` um denselben Faktor erhöht (3.1 Zeile 6), ohne dabei den Aufwand der Funktion zu beeinflussen.
Bei einer Erhöhung der Knotenanzahl wird ebenfalls die Anzahl der Ausführungen von `calculateNextNodeState` um den gleichen Faktor erhöht (3.1 Zeile 4). Allerdings wird dabei auch der Aufwand zur Berechnung der Funktion erhöht, da einmal über die Anzahl der Knoten iteriert wird, um die Nachbarfeatures zu addieren (3.2 Zeile 9).

Die Anzahl der Features beeinflusst nicht die Anzahl an Ausführungen von `calculateNextNodeState`, jedoch ist der Aufwand zur Berechnung der Funktion stark davon abhängig. Es wird mehrmals über Featurevektoren iteriert, um den Durchschnitt der Nachbarfeatures zu berechnen. Allerdings werden in der Funktion `calculateNextNodeState` zweimal Vektoren der Größe `NUM_FEATURES` mit einer Matrix mit der Größe `NUM_FEATURES X NUMFEATURES` multipliziert (3.2 Zeile 25), wobei der Aufwand quadratisch mit der Anzahl an Features zusammenhängt.
Die erwarteten maximalen Beziehungen der Variablen zu der Laufzeit des Benchmarks sind somit:

- `NUM_CALCULATIONS`: proportional
- `NUM_NODES`: quadratisch
- `NUM_FEATURES`: quadratisch

Somit liegt die Laufzeit in $O(T * N * (N + F^2))$ mit der Anzahl an Zeitschritten T, der Anzahl an Knoten N und der Anzahl an Features F.

4 Performancevergleich

In diesem Kapitel werden die Bedingungen, sowie die Laufzeiten des Benchmarks beschrieben. Die Experimentierumgebung wird in Abschnitt 4.1 dokumentiert. Daraufhin folgen die Laufzeiten der Experimente in Abschnitt 4.2. Anschließend gehen wir auf die Einflüsse der Benchmarkparameter in Abschnitt 4.3 ein, um die Performanceunterschiede genauer zu analysieren und mögliche Schwachstellen zu identifizieren.

4.1 Experimentierumgebung

Die Experimente wurden auf einem der NV-Rechner des Fachgebiets Programmiersprachen/-methodik des Fachbereichs 16 der Universität Kassel durchgeführt. Der Rechner besitzt 126GB Hauptspeicher und zwei AMD EPYC 7282 Prozessoren mit jeweils 16 Kernen. Das verwendete Betriebssystem ist Arch Linux. Die Spidroin-Variante wurde mit **g++ 12.2.1** und dem **C++20** Standard kompiliert und nativ auf dem Betriebssystem ausgeführt.
Die Cilk-F-Variante wurde in einem Docker Container ausgeführt. Für Cilk-F wird ein Docker Image bereitgestellt, welches die Verwendung vereinfachen soll, da es alle Abhängigkeiten beinhaltet. Dazu zählt auch eine modifizierte Version von **g++ 5.4.0** welche verwendet wird, um den Benchmark mit dem **C++ 11** Standard zu kompilieren. In der beiliegenden Readme wird gesagt, dass der Overhead für den Docker Container so gering sei, dass kein Unterschied zwischen der Ausführung von Benchmarks im Docker Container und direkt auf dem Host Betriebssystem bemerkbar wäre [3].
Außerdem wurde bei der Kompilierung in beiden Fällen das Flag **-O3** verwendet. Dadurch werden Optimierungen durchgeführt, welche die Ausführungszeit eines

Programms verringern können.

4.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse des Benchmarks mit verschiedenen Parametern gezeigt und diskutiert. Abbildung 4.1 zeigt die Laufzeiten des Benchmarks

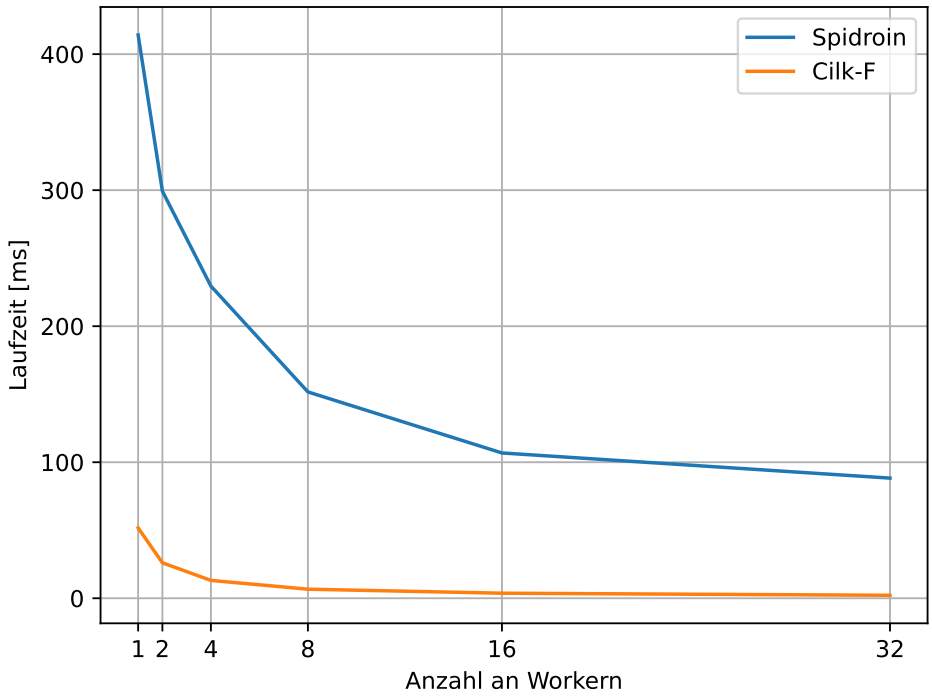

Abbildung 4.1: Laufzeit mit 40 Zeitschritten, 10.000 Knoten und 20 Features

für Cilk-F und Spidroin bei ansteigender Anzahl von Workern. Bei beiden Programmiersystemen ist eine Verringerung der Laufzeit mit einer steigenden Anzahl von Workern zu beobachten, wobei Cilk-F bei jeder Workeranzahl deutlich schneller war als Spidroin.
Auffällig ist die Abweichung der Laufzeiten bei einem verwendeten Worker. Die Laufzeit des Benchmarks mit Spidroin dauerte ca. 410 Sekunden, wohingegen Cilk-F nur

52 Sekunden brauchte. Das bedeutet, Cilk-F war fast achtmal schneller als Spidroin. Dies weist darauf hin, dass die Erzeugung von Futures mit Cilk-F effizienter ist als mit Spidroin.
Jedoch bleibt dieser Faktor mit steigender Workeranzahl nicht konstant.

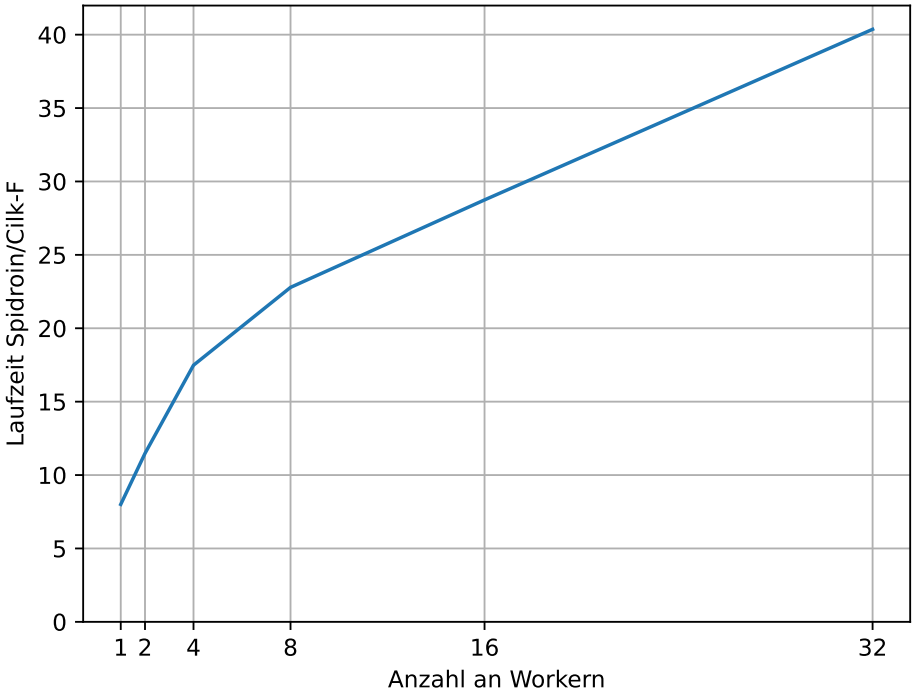

Abbildung 4.2: Verhältnis der Laufzeiten $\frac{t_{Spidroin}}{t_{Cilk-F}}$ mit 40 Zeitschritten, 10.000 Knoten und 20 Features

Abbildung 4.2 zeigt das Verhältnis der Laufzeiten $\frac{t_{Spidroin}}{t_{Cilk-F}}$. Dieses wird mit steigender Anzahl von Workern größer. Das bedeutet, dass Spidroin eine höhere Anzahl von Kernen nicht so effizient nutzt wie Cilk-F. Diese Vermutung wird durch die Betrachtung des Speedups bestätigt.
Abbildung 4.3 zeigt den Speedup der beiden Programmiersysteme sowie den optimalen Speedup. Es ist deutlich zu sehen, dass Spidroin Probleme hat, eine hohe Anzahl von Kernen effizient zu nutzen. Cilk-F zeigt ein besseres Verhalten, weicht

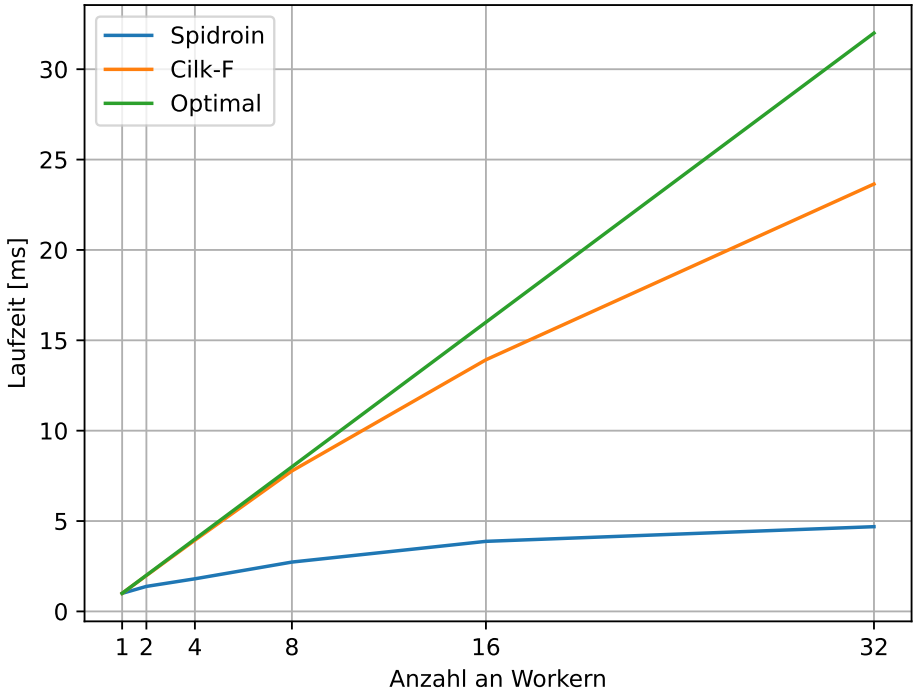

Abbildung 4.3: Speedup $S_n = \frac{T_1}{T_n}$ mit 40 Zeitschritten, 10.000 Knoten und 20 Features

aber auch von der optimalen Linie ab. In beiden Programmiersystemen gibt es einen zusätzlichen Aufwand, welcher mit einer wachsenden Zahl von Workern ansteigt. Dieser zusätzliche Aufwand wird so groß im Verhältnis zur eigentlichen Berechnung eines Futures, dass der Speedup negativ beeinflusst wird.
Durch Erhöhung des Aufwandes zur Berechnung eines Futures kann dieser Einfluss verringert werden. Die Aufwändigkeit der Berechnung eines Futures kann mithilfe der Anzahl von Featuers beeinflusst werden. Auf diese Weise wird die Granularität verringert, weshalb sich der Verwaltungsaufwand ebenfalls verringert.
Die Speedups der Tests mit der doppelten Anzahl an Features sind in Abbildung 4.4 zu sehen. Wie erwartet, verbessert sich der Speedup bei Spidroin bemerkbar, ist aber noch immer weit von der optimalen Linie entfernt. Bei Cilk-F ist nur eine minimale Verbesserung sichtbar, was daran liegt, dass der Verlauf vorher schon deutlich näher

an der optimalen Linie war.

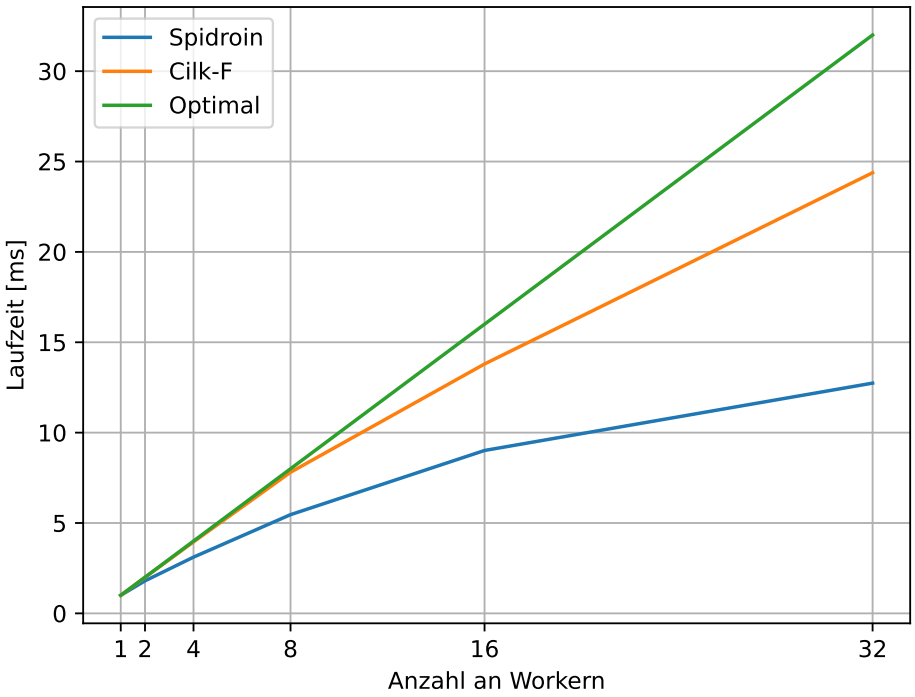

Abbildung 4.4: Speedup $S_n = \frac{T_1}{T_n}$ mit 40 Zeitschritten, 10.000 Knoten und 40 Features

4.3 Einfluss der Benchmark Parameter

In Abschnitt 3.4 wurde abgeschätzt, dass sich die Anzahl an Features quadratisch auf die Laufzeit auswirkt. Das bedeutet, bei einer Verdopplung der Features ist eine Vervierfachung der Laufzeit zu erwarten.

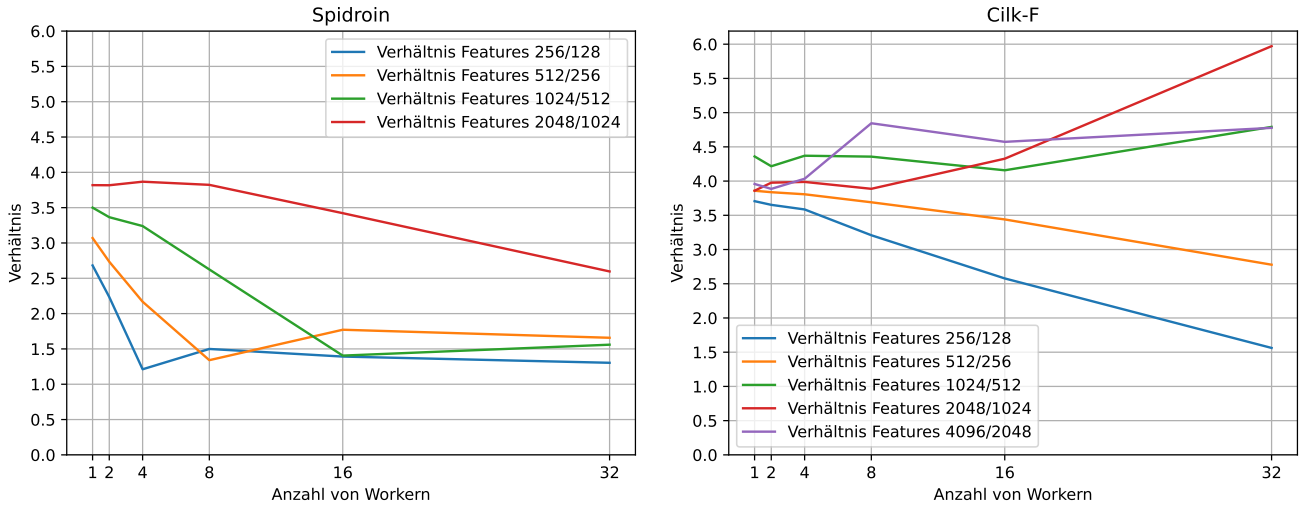

Abbildung 4.5: Verhalten der Laufzeit bei Verdopplung der Features mit 128 Zeitschritten und 128 Knoten

Das Verhalten der Laufzeit bei Verdopplung der Features $\frac{T(2f)}{T(f)}$, wobei $T(f)$ die Laufzeit mit f Features ist, ist in Abbildung 4.5 dargestellt. Im linken Bild sehen wir das Verhalten von Spidroin. Dabei fällt auf, dass sich der Wert mit einer steigenden Anzahl von Features dem Wert 4 annähert. Dies geschieht, da die Berechnungszeit der Futures bei kleineren Anzahlen von Features nicht so stark ins Gewicht fallen. Da Cilk-F die Futures deutlich effizienter verwaltet, sehen wir, dass der Faktor der Laufzeiterhöhung schon bei wenigen Features nah an dem erwarteten Wert vier liegt. Jedoch weichen die Werte bei wachsender Workeranzahl stark ab. Dabei scheint sich das Verhältnis von $\frac{T(4096)}{T(2048)}$ wieder dem Wert von 4 zu nähern. Untersuchungen mit mehr als 32 Kernen wären hier sehr interessant.

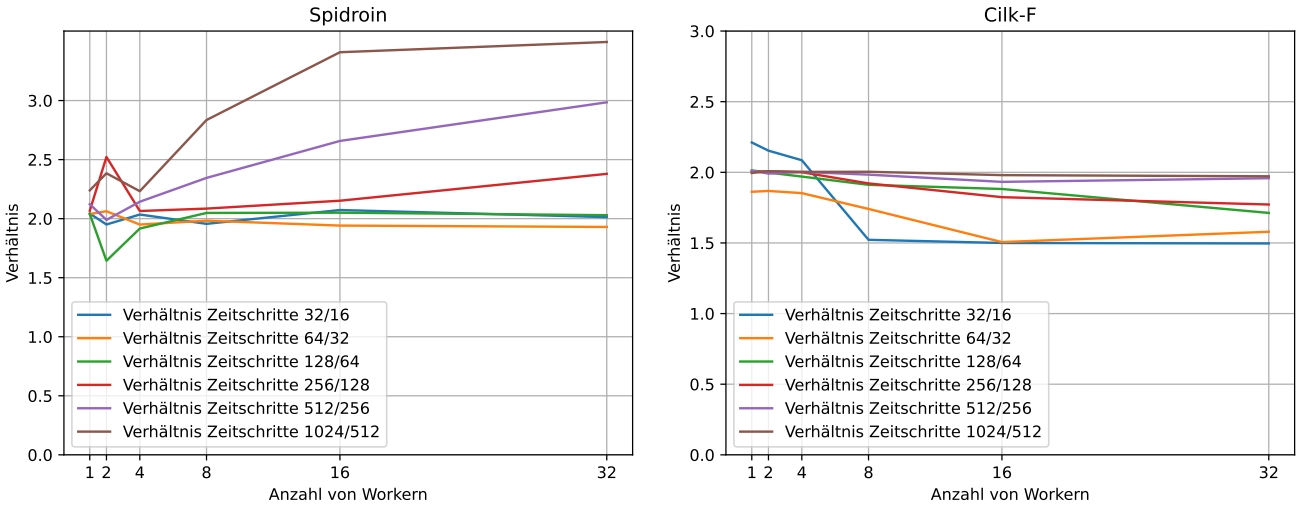

Abbildung 4.6: Verhalten der Laufzeit bei Verdopplung der Zeitschritte mit 128 Knoten und 128 Features

Das Verhältnis zwischen Zeitschritten und Laufzeit wurde als proportional vermutet. Abbildung 4.6 zeigt dieses Verhalten. Im rechten Bild ist das Verhältnis $\frac{T(2t)}{T(t)}$ dargestellt, wobei $T(t)$ die Laufzeit des Benchmarks mit t Zeitschritten ist. Wir können sehen, dass Cilk-F die Erwartung erfüllt, da sich die Laufzeit bei einer Verdopplung der Zeitschritte ebenfalls verdoppelt. Bei wenigen Zeitschritten ist das Verhältnis kleiner als zwei.
Das linke Bild zeigt das gleiche Verhalten für Spidroin. Dort können wir beobachten, dass das Verhältnis mit einer steigenden Anzahl an Zeitschritten wächst.

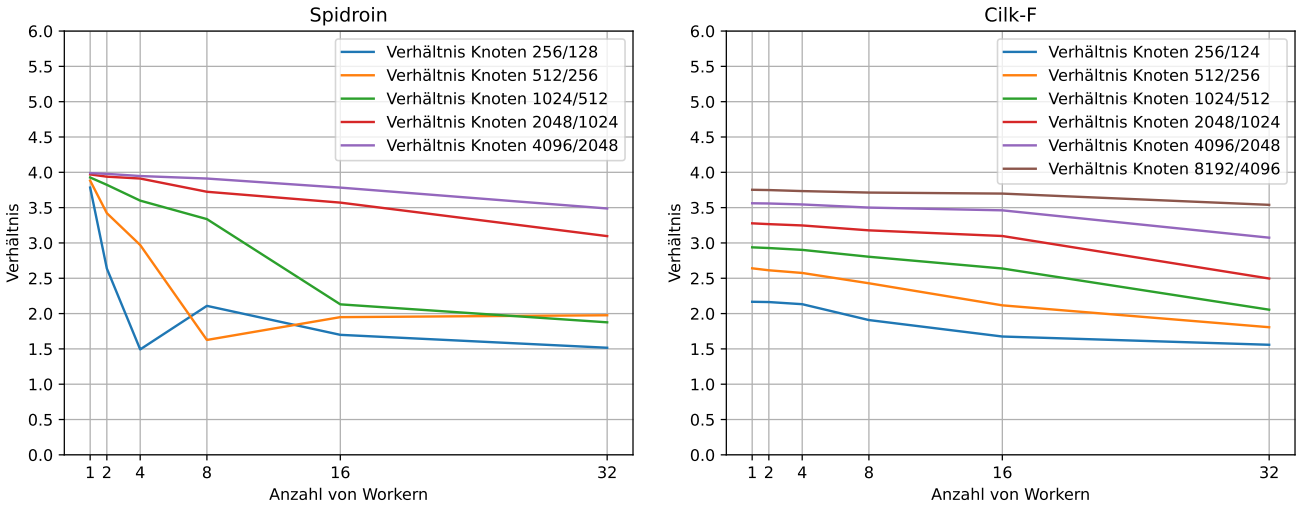

Abbildung 4.7: Verhalten der Laufzeit bei Verdopplung der Knoten mit 32 Zeitschritten und 128 Features

Abbildung 4.7 zeigt das Verhältnis $\frac{T(2n)}{T(n)}$ mit der Laufzeit des Benchmarks $T(n)$ bei n Knoten. Es wird erwartet, dass der Zusammenhang zwischen der Anzahl der Knoten und der Laufzeit quadratisch ist. Aus einer Verdopplung der Knotenanzahl soll eine Vervierfachung der Laufzeit folgen. Das Verhältnis nähert sich in beiden Fällen dem erwarteten Wert an. Bei wenigen Knoten ist der Einfluss einer Verdopplung der Knotenanzahl geringer, da die Laufzeit ebenfalls von den Anzahlen der Zeitschritte und Features beeinflusst wird.
Die Auswirkungen der Anzahlen von Zeitschritten und Knoten sind auffällig, da sich diese unterscheiden. Bei der Knotenanzahl verhalten sich Spidroin und Cilk-F gleich, aber bei der Anzahl der Zeitschritte unterscheiden sie sich. Der Faktor vier wird bei der Verdopplung der Knotenanzahl erreicht, welcher aus einer Verdopplung der Futures und einer Verdopplung des Aufwandes zur Berechnung eines Futures entsteht. Allerdings wird der Faktor zwei bei der Verdopplung der Zeitschritte mit Spidroin übertroffen, obwohl dieser aus einer Verdopplung der Futures resultiert. Jedoch unterscheiden sich die Verdopplungen der Futures, welche aus den Verdopplungen der Knotenanzahl und Zeitschritten erfolgen. Bei einer Erhöhung der Knotenanzahl werden mehr Futures erzeugt, welche im selben Zeitschritt existieren und daher

parallel berechnet werden können, ohne aufeinander warten zu müssen. Bei der Erhöhung der Zeitschritte werden mehr Futures in neuen Zeitschritten erzeugt. Dies hat zur Folge, dass bei der Erhöhung der Knotenanzahl öfter auf den Wert eines Futures gewartet werden muss, was bei der Erhöhung der Zeitschritte nicht der Fall ist. Das könnte bedeuten, dass Spidroin sich weniger effizient verhält, wenn auf Futures gewartet werden muss.

5 Fazit

Abschließend kann die Frage zum Speedup und der Laufzeit im Vergleich zu Cilk-F beantwortet werden. In den Experimenten hat sich gezeigt, dass Cilk-F effizienter ist als Spidroin. Die Laufzeiten unterscheiden sich um den Faktor acht bei einem Worker bis zu über 40 bei 32 Workern .
Der Speedup ist schwächer als bei Cilk-F.
Außerdem beeinflussen die Anzahl der Futures und der Aufwand zur Berechnung der einzelnen Futures die Effizienz und Skalierbarkeit stärker, als es bei Cilk-F der Fall ist. Gründe dafür könnten sein, dass die Erzeugung und Verwaltung von Futures, sowie das Warten auf Futures in Spidroin weniger effizient sind.
Um die Vermutungen weiter zu untersuchen, wären Testläufe mit mehr als 32 Workern und größeren Benchmarkparametern interessant. Es wurde auch an einer Version gearbeitet, bei der nicht alle Futures zu Beginn des Benchmarks gespawnt werden, sondern die Futures in Zeitschritten $t > 1$ von ihrem direkten Vorgänger gespawnt werden. Allerdings wurde dieser Ansatz aufgrund des Umfangs der Arbeit nicht weiter verfolgt.
Die Verwendung von Spidroin hat sich bei der Implementierung der Benchmarks als einfacher herausgestellt, da im Gegensatz zu Cilk-F keine Makros benötigt werden. Durch die häufige Notwendigkeit der Makros in Cilk-F kam es bei der Entwicklung zu vielen Speicherzugriffsfehlern, welche schwierig zu identifizieren waren. Des Weiteren schiebt Cilk-F viele Variablen im Speicher herum, weshalb mit der Sichtbarkeit von Variablen aufgepasst werden muss und Futures nicht alle Typen beinhalten können.
Die Installation beider Programmiersysteme war problemlos. Dies war bei Cilk-F jedoch nur durch den bereitgestellten Docker Container möglich, was ein Nachteil ist, falls dieser nicht verwendet werden soll.

Literaturverzeichnis

[1] Thilo Bayer - PCGH. *Zeitreise: Entwicklung der CPU-Kerne von 2015 bis heute*. URL: https://www.pcgameshardware.de/CPU-CPU-154106/News/Zeitreise-Entwicklung-der-CPU-Kerne-von-2015-bis-heute-1321063/ (besucht am 23. 08. 2023).

[2] Yifan Xu Kyle Singer und Angelina Lee. „Proactive Work Stealing for Futures“. In: *PPoPP '19: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 2019, S. 257–271. DOI: 10.1145/3293883.3295735.

[3] Kyle Singer, Yifan Xu und I-Ting Angelina Lee. *Proactive Work Stealing for Futures: Cilk-F*. Version 1.0.1. Funding for this work was also provided by National Science Foundation (US) grant ÄITF: Applied Algorithmic Foundation for Scheduling Multiprogrammed Parallelizable Workloads"(1733873). Dez. 2018. DOI: 10.5281/zenodo.2227457. URL: https://doi.org/10.5281/zenodo.2227457.

[4] Giancarlo Zaccone. *Python parallel programming cookbook*. Bd. 2. Packt Publishing, 2019, 29f. ISBN: 9781789530063.

[5] Heller T. et al. Thoman P. Dichev K. „A taxonomy of task-based parallel programming technologies for high-performance computing“. In: *The Journal of Supercomputing* 3 (2018), S. 1422–1434. DOI: 10.1007/s11227-018-2238-4.

[6] CPPReference. *C++ Coroutines*. URL: https://en.cppreference.com/w/cpp/language/coroutines (besucht am 23. 08. 2023).

[7] Daniel Sonnet. *Neuronale Netze kompakt*. Bd. 1. Springer Fachmedien Wiesbaden, 2022, S. 18–20. ISBN: 9783658290818.

[8] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce und Alexander B. Wiltschko. „A Gentle Introduction to Graph Neural Networks“. In: *Distill* (2021). https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.