Automatic HICANN init and script performance optimizations

Experiment Performance Improvements

Patrick Häussermann

7. September 2018

Abstract

This report details the results of a small project given in preparation for an upcoming bachelor thesis at the *BrainScaleS* project at Heidelberg University.
Within the scope of this project significant speed improvements were made to scripts preceding most experiment runs and options for the automated initialization of HICANNs were added.

Zusammenfassung

Diese Arbeit beschreibt die Ergebnisse eines Praktikums, welches ihm Rahmen einer angehenden Bachelorarbeit am *BrainScaleS* Projekt der Universität Heidelberg absolviert wurde.
Es wurden signifikante Geschwindigkeitsverbesserungen an Skripten gemacht, die während eines Experimentablaufs automatisch gestartet werden. Zusätzlich wurden erste Funktionen implementiert, welche die Lizenzverwaltung sowie die automatische Initialisierung von HICANNs erlauben.

1 Einleitung

Der Kern des aktuellen BrainScaleS Aufbaus besteht aus 20 Wafermodulen verteilt auf fünf Standard 19" Netzwerkschränke. Jedes Wafermodul wird dabei in 48 sog. "reticles" unterteilt, wobei jedes reticle 8 HICANNs (*High-Input Count Analog Neuronal Network Chip*) enthält, welche für die Modellierung der Neuronen und Synapsen zuständig sind. Zusätzlich ist jedem dieser reticle ein FPGA zugeordnet, welcher für die Kommunikation mit dem Rest des Aufbaus zuständig ist.

Unterstützt werden die Wafermodule durch 20 *compute nodes*, wovon jeder während der Konfiguration und Experimentabläufe die gebündelte Kommunikation mit bis zu 48 FPGAs steuern kann. Für hochverfügbaren Netzwerkspeicher, auf den alle *nodes* zugreifen können, stehen vier weitere *storage nodes* zur Verfügung.

2 Slurm workload manager

Um die faire und effiziente Verteilung der Jobs und Experimente auf die vorhandenen Ressourcen zu garantieren, kommt ein sog. *job scheduler* zum Einsatz. Das *BrainScaleS*-Projekt setzt dabei auf eine den speziellen Bedürfnissen angepasste Version des *slurm workload manager*[1]. Dabei handelt es sich um ein open-source Projekt, welches auch auf vielen Supercomputern zum Einsatz kommt.

Dieser *scheduler* stellt zudem sicher, dass Ressourcen exklusiv einem Nutzer zugeordnet werden können. So kann verhindert werden, dass zwei oder mehr Experimente die selbe Ressource ansprechen und sich gegenseitig beeinflussen. Zudem ist es möglich neue *jobs* entweder direkt zu starten, oder per Stapelverarbeitung *(batch mode)* abarbeiten zu lassen.

2.1 Lizenzverwaltung

Ein Teil dieser Arbeit nutzt eine bereits implementierte Erweiterung von *slurm*, welche die Verwaltung der Hardwareressourcen in Form von Lizenzen erlaubt.
Slurm bietet zwar bereits die Möglichkeit zur Verwaltung von Lizenzen, allerdings ist dies nur für reine Softwarelizenzen gedacht. Um auch Hardwarelizenzen verwalten zu können, musste für jede vom Nutzer ansprechbare Hardware eine Lizenz erstellt werden. Da der Nutzer zudem weitreichende Kontrolle über die von seinem Experiment genutzte Ressourcen hat, musste *slurm* um ein zusätzliches `job sumbit` Plugin erweitert werden. Dieses Plugin erkennt unter anderem die vom Nutzer übergebenen Parameter und stellt die dazugehörigen Lizenzen und Informationen in Form von *Umgebungsvariablen* zur Verfügung.

Um Lizenzen verwalten zu können, erwartet *slurm* eine Datei, in der alle verfügbaren Lizenzen hinterlegt sind. Eine der Verbesserungen, die im Rahmen dieser Arbeit gemacht wurde, stellt ein kleines Pythonskript dar, welches diese Lizenzdatei automatisch durch Auslesen der Hardwaredatenbank erstellen kann. Durch Einbindung in den CI/CD Ablauf mit *Jenkins* kann sichergestellt werden, dass die Lizenzen immer dem aktuellen Stand der Hardware entsprechen. Aktuell umfasst diese Lizenzdatei etwa 1000 Lizenzen[2].

[1] `https://slurm.schedmd.com/overview.html`
[2] Am 27.07.2018 erstellte Lizenzdatei

3 Prolog und Epilog Skripte

Eines der Ziele dieser Arbeit war die Optimierung einiger Bashskripte, welche von *slurm* automatisch an vordefinierten Punkten eines Jobverlaufs gestartet werden können.
Grundsätzlich kennt und unterstützt *slurm* mehrere solcher Möglichkeiten zum Skript- bzw. Programmstart. In der aktuellen Konfiguration werden davon jedoch nur Prolog und Epilog, sowie Taskprolog und Taskepilog genutzt.
Eine Übersicht wann und in welchem Kontext diese Skripte laufen, kann der offiziellen Dokumentation entnommen werden [3].

3.1 prolog.sh

Das Prolog-Skript läuft standardmäßig direkt bevor der erste Schritt in der Abarbeitung eines Jobs ausgeführt wird.

Die aktuelle Aufgabe dieses Skripts besteht hauptsächlich darin spezifische `iptable`-Regeln zu setzen, um die Netzwerkkommunikation mit dem/den FPGA(s) während eines Experimentablaufs zu steuern.
Die Erkennung der genutzten FPGA(s) erfolgt durch Auslesen der Lizenzen aus den Umgebungsvariablen (siehe 2.1) und Berechnung der spezifischen FPGA-Parameter wie IP- und MAC-Adresse.

3.2 epilog.sh

Das Epilog-Skript läuft grundsätzlich nach Beendigung jedes Jobs.

Genutzt wird dieses, um die Änderungen an der Netzwerkkommunikation rückgängig zu machen, die durch das Prolog-Skript vorgenommen wurden. Die Kommunikation mit FPGA(s) muss nur während eines Experimentablaufs gesteuert werden.

3.3 Taskepilog.sh und Taskprolog.sh

Diese Skripte laufen nur mit den rechten des entsprechenden Users und sind aktuell nur für das Logging einiger Parameter, sowie dem Löschen etwaiger *shared memory files* zuständig. Diese können nach dem vorzeitigen Abbruch eines Experiments auf dem System verbleiben.

4 Lizenzverwaltung und automatische Initialisierung

Gegenstand des Praktikums war auch die Bereitstellung von Funktionen, welche die Lizenzverwaltung per slurm-eigener Datenbank erlauben, sowie darauf aufbauend die

[3] `https://slurm.schedmd.com/prolog_epilog.html`

Implementierung einer automatische Initialisierung von Lizenzen.

Die aktuelle Installation ist so konfiguriert, dass Informationen über aktuelle und abgeschlossene Jobs in einer MySQL-Datenbank gespeichert werden. So können unter anderem sehr einfach Berichte und Nutzungsstatistiken erstellt werden.

Diese Datenbank wurde genutzt, um die Skripte mit Funktionen zu erweitern, die es erlauben den Status der Lizenzen (oder genauer gesagt: den Initialisierungszustand der zugrunde liegenden HICANNs) zu verfolgen. Dabei sind immer 8 HICANNs einem reticle bzw. FPGA und damit einer Lizenz zugeordnet.
Für die automatische Initialisierung mussten sowohl das `job_submit` Plugin als auch das `prolog.sh` Skript weiter angepasst und im Funktionsumfang erweitert werden.

5 Verbesserungen

Alle Skripte wurden sowohl auf die Ausführungsgeschwindigkeit hin optimiert, als auch im Funktionsumfang erweitert.
Dabei konnte die Ausführungsgeschwindigkeit zum Teil drastisch verbessert werden (siehe 6). Die größten Änderungen fanden an `prolog.sh` statt, welches bereits um einige Funktionen erweitert wurde, die in einer angehenden Bachelorarbeit nochmals zusätzlich ausgebaut werden sollen.

Funktionen, die in mehr als einem Skript vorkommen, wurden in eine eigene Datei ausgelagert. Ebenfalls wurde die Möglichkeit für *unit tests* durch ein eigenständiges Skript bereitgestellt. Dies verbessert die Übersichtlichkeit und erlaubt es spätere Ergänzungen einfacher zu implementieren und zu testen.

5.1 Geschwindigkeitsverbesserungen

Die Geschwindigkeitsverbesserungen wurden hauptsächlich durch die Nutzung von *Bashisms* erreicht. Dabei handelt es sich um spezielle Programme und Funktionen der *Bash Shell*, welche heutzutage die Standard-Shell auf vielen Linuxdistributionen darstellt.
Diese *Bashisms* bieten gegenüber den allgemein definierten POSIX-Standards große Vorteile in Sachen Geschwindigkeit und Funktionsumfang.

Der Ablauf aller Schleifen und die darin stattfindende Lizenzerkennung wurden ebenfalls optimiert. Die Berechnung der FPGA Parameter konnte durch die ausschließliche Verwendung von Bash-eigenen Funktionen beschleunigt werden.
Allgemein konnte der Skriptablauf durch den Verzicht auf *Pipes* signifikant beschleunigt werden.

5.2 Lizenzverwaltung

Das `prolog.sh` Skript wurde um Funktionen erweitert, welche die Verwaltung von Lizenzen in einer slurm-eigenen Datenbank erlauben. Durch die Nutzung von *prepared statements* können damit schnell und einfach Einträge gesetzt und abgerufen werden.

Aktuell werden in der MySQL Datenbank `slurmdbd.slurmviz_license_data` folgende Einträge verwaltet:

```
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| license   | varchar(15)  | NO   | PRI |         |       |
| status    | varchar(255) | YES  |     | NULL    |       |
| init_time | datetime     | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

- **`license`** Lizenzname wie er *slurm* aus der Lizenzdatei bekannt ist, z.B.: `W14F15`. Siehe auch 2.1.
- **`status`** Status der Lizenz. Entweder `'CLEAN'` oder `'DIRTY'`, je nach Initialisierungszustand der Lizenz.
- **`init_time`** Datum der Initialisierung in der Form: `YYYY-MM-DD HH:MM:SS` bzw. `'NULL'` für Lizenzen mit Status `DIRTY`.

5.3 Automatische Initalisierung

In Zukunft soll die automatische Initialisierung der Nachbarlizenzen das Standardverhalten bei jedem Experimentstart werden.
Dies ist wichtig, da auch die dem Experiment benachbarten HICANNs vor Experimentstart initialisiert werden sollten, um einen störungsfreien Ablauf zu gewährleisten.

Dabei sollen jedoch nur HICANNs bzw. Nachbarlizenzen initialisiert werden, welche **(a)** als `DIRTY` in der Datenbank markiert sind, oder **(b)** deren Initialisierungsdatum länger als ein Tag zurück liegt.
Die Ermittlung der dem Experiment benachbarten Lizenzen erfolgt dabei durch das `job_submit` Plugin. Dieses wurde um zusätzliche Funktionalität und zwei optionale Parameter erweitert und ermittelt nun automatisch alle dem Experiment benachbarten Lizenzen. Diese Nachbarlizenzen werden in eine Umgebungsvariable `$NEIGHBOUR_LICENSES` exportiert. Sollte zusätzlich einer der optionalen Parameter spezifiziert worden sein, wird dieser ebenfalls in `$SLURM_HICANN_INIT` zur Verfügung gestellt.

Dies ist nötig, da die letztendliche Bestimmung und Initialisierung der Lizenzen durch das `prolog.sh` Skript erfolgt. Dieses unterscheidet je nach Inhalt von `$SLURM_HICANN_INIT` zwischen einem von drei möglichen Fällen (siehe Tabelle 1). Die Initialisierung der Lizenzen soll durch den Aufruf einer externen *Init-binary* erfolgen. Diese kann z.B. direkt aus

dem Ablauf des `prolog.sh` Skripts heraus gestartet werden, wobei die Lizenzen direkt als Parameter übergeben, oder aus der Umgebungsvariablen ausgelesen werden können.

Wie bereits erwähnt sollen standardmäßig nur Lizenzen initialisiert werden, welche sich in `$NEIGHBOUR_LICENSES` befinden und nach Abfrage der Datenbank als "DIRTY" zurück gegeben werden (Fall 3).
Wird vom Nutzer einer der optionalen Parameter angeben, wird das Standardverhalten überschrieben und es werden entweder alle (`--force-hicann-init`) Lizenzen initialisiert, unabhängig deren Status (Fall 2), oder die Initialisierung wird übersprungen (`--skip-hicann-init`, Fall 1).
Es kann jeweils nur einer der beiden Parameter angeben werden. Eine Übersicht über alle möglichen Abläufe findet sich in nachfolgender Tabelle 1:

Fall	**Parameter**	**Inhalt der Umgebungsvariable**		**Auswirkung**
		`$SLURM_HICANN_INIT`	`$NEIGHBOUR_LICENSES`	
1	`--skip-hicann-init`	`"SKIP"`	**null** (leer)	Überspringt Initialisierung komplett.
2	`--force-hicann-init`	`"FORCE"`	Alle dem Experiment benachbarten Lizenzen (kommagetrennt)	Initialisierung **aller** Nachbarlizenzen
3	Kein Parameter angegeben	**null** (wird nicht gesetzt)		Automatische Initalisierung der als "DIRTY" hinterlegten Lizenzen.

Tabelle 1: Optionale Parameter zur Steuerung der automatischen Initialisierung (Fall 1 und 2), sowie das Standardverhalten (Fall 3).

Die bereits angesprochene Umgebungsvariable `$NEIGHBOUR_LICENSES` enthält für die Fälle **(2)** und **(3)** die Lizenzen der dem Experiment benachbarten HICANNs bzw. FPGAs. Soll die automatische Initialisierung übersprungen werden (Fall **1**), ist diese leer.

Abbildung 1 zeigt den Ablauf der automatischen Initialisierung. Es gilt zu beachten, dass dort nur Schritte aufgeführt sind, welche für die Initialisierung relevant sind. Dies gilt besonders für den Ablauf innerhalb des Prolog-Skripts, welches in Realität unter anderem auch die Netzwerkkommunikation während des Experiments steuert (siehe 3.1).

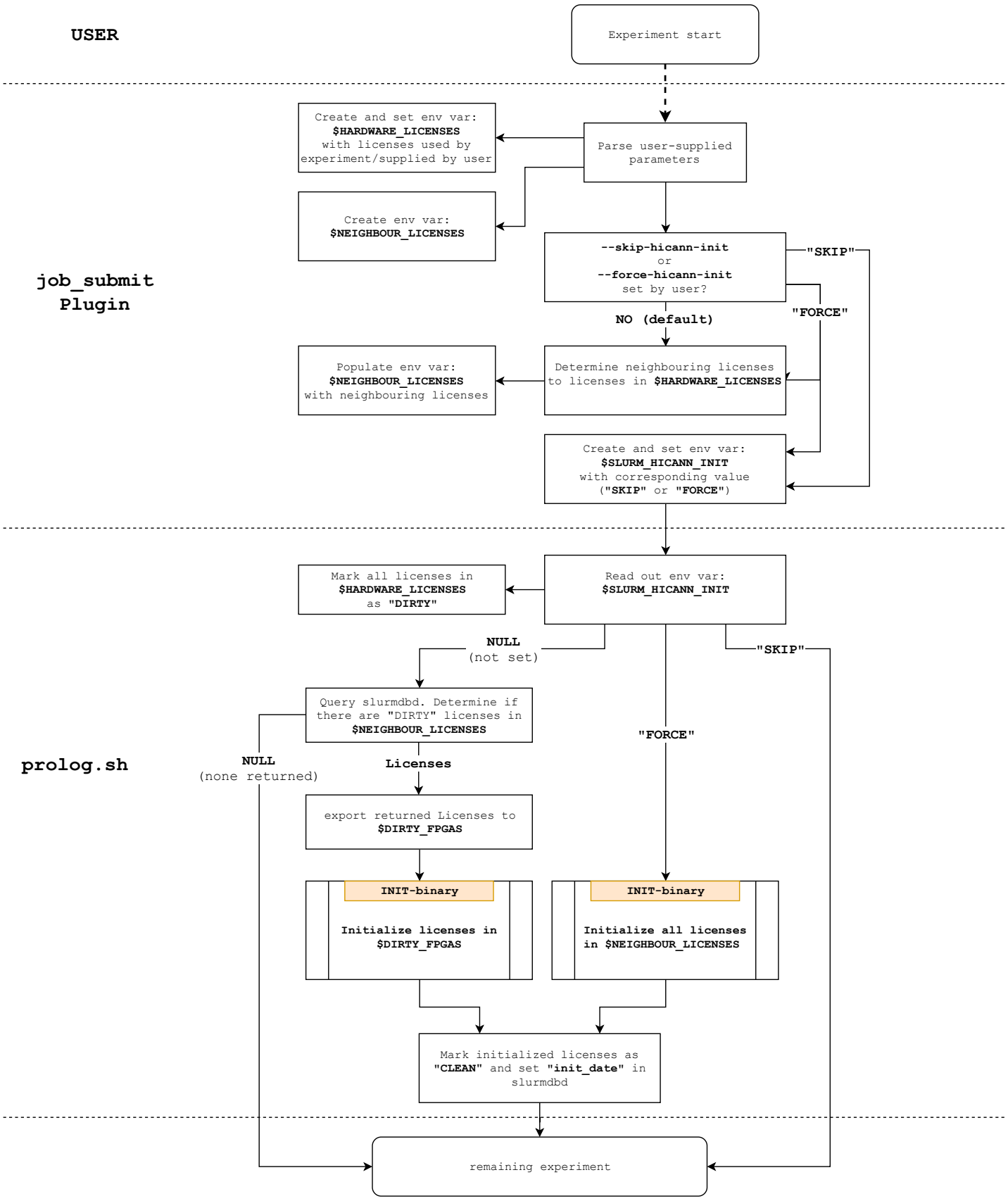

Abbildung 1: Vereinfachter Ablauf der automatischen Initialisierung. Aufgeführt sind nur die für die Initialisierung relevanten Schritte.

6 Messungen

Alle Geschwindigkeitsmessungen wurden auf einem Xeon E3-1231 v3 mit 16 Gb RAM und NVME-SSD durchgeführt. Getestet wurde die Ausführungsgeschwindigkeit mit 1, 5, 10 sowie 144 Lizenzen.

Messungen erfolgten mit dem Bash-eigenen Programm `time`. Gestoppt wurde die Zeit für je 100 Ausführungen des wichtigen Prolog-Skripts[4] (`prolog.sh`). Die angegebene Zeit entspricht dabei der Realzeit, also der vergangenen Zeit vom Start bis zum Ende des Durchlaufs.[5]

	Laufzeit (s)		
# Lizenzen	**optimiert**	**unoptimiert**	**Verbesserung (%)**
1	0,990	2,156	218
5	1,285	6,543	509
10	1,799	12,311	684
144	14,650	171,365	> 1000

Tabelle 2: Vergleich der Ausführungsgeschwindigkeit für je 100 Durchläufe des neuen (optimierten) Skripts mit dem Alten (unoptimierten).

Es lässt sich feststellen, dass die optimierten Skripte selbst unter Verwendung nur einer Lizenz bereits mehr als doppelt so schnell laufen und ihren Vorsprung ausbauen können, je mehr Lizenzen verwendet werden.
Dies lässt sich vor allem auf die Optimierung der Schleifen, sowie der darin stattfindenden Lizenzerkennung zurückführen (siehe 5.1). Eigene Tests ergaben, dass der in der optimierten Version verwendete `for-loop` etwa doppelt so schnell wie das zuvor verwendete `while-read` Konstrukt arbeitet.
Aber auch in der Lizenzerkennung bzw. der darauf aufbauenden Berechnung der FPGA-Parameter konnte durch die Nutzung von *Bashisms* und dem Verzicht auf *Pipes* die Ausführungsgeschwindigkeit deutlich verbessert werden.

Die hier getestete Maximalanzahl von 144 Lizenzen (entspricht 3 vollen Wafer) ist zwar unrealistisch, belegt aber den Trend, wonach die optimierten Skripte ihren Geschwindigkeitsvorsprung ausbauen können, je mehr Lizenzen verwendet werden.

[4] Für alle Messungen wurde in den getesteten Skripten Zeilen auskommentiert, die Änderungen am Netzwerk veranlassen (z.B. durch Aufruf von `iptables` siehe 3.1). Diese weisen nur geringe Unterschiede zwischen beiden getesteten Versionen auf. Schleifen, die dadurch gegebenenfalls leer wurden, wurden durch Einfügen eines ":" Charakters trotz fehlendem Inhalt zum Ausführen gezwungen.

[5] Die hier gemessenen Zeiten variieren je nach Auslastung des Systems. In wiederholten Messungen konnte aber keine Abweichung größer als 6% festgestellt werden.

7 Zusammenfassung

Im Rahmen des Praktikums wurden Optimierungen der von *slurm* automatisch gestarteten Skripte vorgenommen. Durch Änderung des Skriptablaufs, sowie Verwendung von Shell-eigenen Programmen und Funktionen konnte eine signifikante Geschwindigkeitsverbesserung erreicht werden. Im Falle des Prolog-Skripts, welches die meisten Änderungen und Anpassungen erfuhr, konnte die Ausführungsgeschwindigkeit um mindestens 200% erhöht werden (6).

Ebenfalls wurde der Funktionsumfang stark ausgebaut. Die Nutzung von *prepared statements* erweitert das bestehende Skript um die Möglichkeit Hardwarelizenzen in einer slurm-eigenen Datenbank (5.2) zu verwalten. Ein zusätzliches Pythonskript ermöglicht es Lizenzen automatisch durch Auslesen der Hardwaredatenbank zu erstellen. Dieses kann in den aktuellen CI/CD Ablauf integriert werden, um bei Änderungen der Hardwaredatenbank die von *Slurm* verwalteten Lizenzen auf dem aktuellen Stand der Hardware zu halten.

Zusammen mit der erweiterten Funktionalität des `job_submit` Plugins (5.3) erlauben diese Änderungen nun die automatische Initialisierung der dem Experiment benachbarten Lizenzen, sowie die Steuerung der Initialisierung durch den Nutzer.
Nach weiteren Tests soll die automatische Initialisierung der benachbarten Lizenzen das Standardverhalten bei jedem Experimentablauf werden.

Glossar

Pipe Gepufferter Datenstrom zwischen zwei Prozessen nach FiFo Prinzip. Die Nutzung von anonymen Pipes in Unix geschieht durch das Verbinden zweier Prozesse mit dem "|"-Charakter. 5, 8

POSIX Das "Portable Operating System Interface " stellt eine standarisierte Programmierschnittstelle zwischen Anwendungssoftware und Betriebssystem dar. 4

prepared statement "Vorbereitete Anweisung" welche zuerst Platzhalter statt Parameterwerte enthält. Kann das Ausnützen von Sicherheitslücken verhindern und bietet Geschwindigkeitsvorteile bei mehrmaliger Ausführung. 5, 9