Praktikumsbericht

von Stephen Schaumann

Zeitraum: 01.09.2017 - 31.10.2017

Spikey in Multi-User Umgebungen

Inhaltsverzeichnis

1 Ziel

Bevor meinem Praktikum war der Zustand des Spikey so, dass er zu jedem gegebenen Zeitpunkt nur von einem einzigen Nutzer verwendet werden konnte und Programme für den Spikey manuell per SLURM gestartet werden mussten.
Am Anfang eines Skripts für den Spikey wird in der Funktion *PyNN.setup()* sofort eine Verbindung zum Chip hergestellt, welche diesen für andere Nutzer blockiert. Diese Verbindung bleibt während dem gesamten Programm bestehen, ab *PyNN.setup()* bis zum Ende des PyNN-Skriptes durch *PyNN.end()*, obwohl sie nur für einen Teil davon, in dem das Experiment tatsächlich stattfindet, benötigt wird.
Ziel des Praktikums war es, diese Verbindung zum Spikey erst dann herzustellen wenn sie benötigt wird und direkt zu löschen wenn sie nicht mehr verwendet wird.

2 Arbeitsablauf im Programm

Im Folgenden werden die einzelnen Schritte in der Nutzung des Spikey genauer erläutert, vor allem hinsichtlich ihrer Bedeutung für die Verbindung zur Hardware.
Experimente für den Spikey werden in Python-Skripten spezifiziert. Mithilfe des Pakets *PyNN* können die nötigen Objekte erstellt und konfiguriert werden und an den Chip geschickt werden.
Zunächst werden mit *PyNN.setup()* grundlegende Variablen definiert. Diverse Werte stehen in *_hardwareParameters*, wie die spikeyNr, welche Version der Spikey ist, wie hoch der Beschleunigungsfaktor ist, etc. Mithilfe von *hardware.initialize()* werden u.a. die Hardware-Objekte "net"(hierin wird das Netzwerk an Neuronen gespeichert) und "hwa"("Hardware Access") erzeugt. Direkt im Anschluss wird durch *hardware.getBus()* die Verbindung zur Hardware etabliert und der workStationName durch diese Verbindung ausgelesen. workStationName ist ein String der die genaue Station identifiziert, auf der sich der verwendete Spikey befindet. Dieser Name ist außerdem nötig um die Default-Einstellungen für diesen Spikey zu laden, die nach workStationName geordnet in einer Default-Config Datei gespeichert sind. Zuletzt wird mit *hardware.getHardware()* das hwa-Objekt konfiguriert und die Hardware bereit zur Nutzung gemacht.
Nachdem *PyNN.setup()* aufgerufen wurde findet der Großteil des Experiments statt: das Netzwerk von Neuronen und Synapsen wird erstellt und konfiguriert, Parameter für das Experiment werden eingestellt (z.B. welche Daten gespeichert werden sollen), und sonstige Optionen wie STP (short-term plasticity) oder STDP (spike timing dependent plasticity) können aktiviert/spezifiziert werden.
Wenn dieser Hauptteil des Programms abgeschlossen ist wird das Experiment per *PyNN.run()* durchgeführt. Das zuvor erstellte Netzwerk wird darin in Hardware-Koordinaten umgewandelt (*hardware.mapNetworkToHardware*). Diese gemapten Daten werden dann an den Spikey gesendet und das Experiment laufen gelassen. Die Ergebnisse werden dann, wie zuvor spezifiziert, gespeichert.
Danach können diese Daten nach Belieben analysiert und geplottet werden. Die Objekte des Experiments sind alle noch vorhanden, bis die Funktion *PyNN.end()* aufgerufen wird. Hierin werden die Hardware-Objekte gelöscht mithilfe von *hardware.delHardware()* und somit die Verbindung zum Spikey terminiert. Ab diesem Punkt ist die Nutzung des Spikey beendet und es wird höchstens noch lokal mit den Resultaten gearbeitet.

3 Ablauf der Verbesserungen

3.1 Zentralisierung der Hardwarenutzung

Zunächst wurde jeglicher Zugriff auf die Hardware auf die Funktion *PyNN.run()* beschränkt.
Der workStationName wurde daher nicht mehr durch die Verbindung zur Hardware ausgelesen, sondern direkt aus der Umgebung ausgelesen durch
"os.environ['MY_STAGE1_STATION']". Da per Slurm alle Befehle direkt an einen Cluster-Knoten geschickt werden müssen ist nämlich sowieso bekannt auf welcher Station man arbeitet, daher ist Nutzung des Chips hier unnötig.
Eine große Änderung war das Aufteilen der Funktion *hardware.getHardware()* in zwei neue Funktionen: *loadCalibration()* und *hardware.initHardware()*. Der Code selbst wurde dabei nicht signifikant verändert sondern lediglich auf die beiden neuen Funktionen verteilt. *loadCalibration()* lädt, wie der Name sagt, die Kalibrationen für den Chip, etwa die Anzahl verfügbarer Neuronen/Synapsen, etc.. Diese Funktion wird weiterhin in *PyNN.setup()* aufgerufen und ermöglicht das weitere arbeiten, allerdings ohne eine Verbindung zum Spikey herzustellen.
Die tatsächliche Verbindung wird nun erst in *hardware.initHardware()* hergestellt, welche aber ruhig später aufgerufen werden kann. Diese Funktion wurde daher an den Anfang von *PyNN.run()* verschoben.
Als letztes musste das Löschen der Verbindung hervorgezogen werden von *PyNN.end()*. Die darin verwendete Funkton *hardware.delHardware()* ruft zunächst
hardware.hwa.delHardware() auf und löscht danach die Hardware-Objekte hwa und net. Die zweite Funktion, *hardware.hwa.delHardware()* löscht jedoch bereits die Verbindung,

die innerhalb des Objektes hwa stattfindet. Interessante Objekte, wie die Spiketrains von dem Experiment, werden dadurch jedoch nicht beeinflusst. Daher kann man diese Funktion bereits am Ende von *PyNN.run()* aufrufen ohne den weiteren Ablauf des Programms zu stören. Damit war nun sämtliche Nutzung des Chips auf die Funktion *PyNN.run()* beschränkt.

3.2 Auslagerung in Datei

Als Vorbereitung für die weiteren Schritte wurde der Teil von *PyNN.run()*, der den Spikey direkt nutzt, in eine seperate Datei ausgelagert. Diese konnte an der entsprechenden Stelle als Subprozess aufgerufen werden. Alle für den run benötigten Dateien wurden dafür in ein dictionary "run_data" verpackt, darunter _hardwareParameters, die Hardware-Objekte hwa und net als eigenes dictionary "hardwareObjects", und diverse Variablen. Vor dem Start des Subprozess wurde "run_data" mithilfe von pickle in einer Datei verpackt, welche dann am Anfang der seperaten Datei ebenfalls mit pickle wieder entpackt werden konnte. Hierfür musste für einige Objekte picklen ermöglicht werden, da dies für kompliziertere Objekte nicht automatisch möglich ist. Besonders hwa und net waren dabei eine Herausforderung, da sie eine komplizierte Struktur mit vielen weiteren Objekten beinhalten. Beispielsweise die Klasse PySpikeTrain musste hierfür speziell bearbeitet werden. In der __init__-Datei für den Spikey wurden daher nachträglich Methoden *SpikeTrain_getState()* und *SpikeTrain_setState()* definiert und der Klasse in pyhal_c_interface_s1v2 zugewiesen.
Die ebenfalls dort zu findende Klasse PySpikeyConfig konnte nicht so gut zum picklen ermöglicht werden. Sie beinhaltet jedoch bereits eine Methode um die wichtigen Informationen in eine Datei zu schreiben, was standardmäßig gemacht wird um nach einem Experiment die Konfiguration des Spikey anschauen zu können. Mithilfe dieser Methode lässt sich vor dem Subprozess das Objekt speichern und kann dann im Subprozess wieder eingelesen werden.
Nachdem nun alles durch pickle verpackt werden konnte oder anderweitig übertragen wurde lief der Subprozess sauber ab und vereinte in sich die gesamte Nutzung des Spikey.

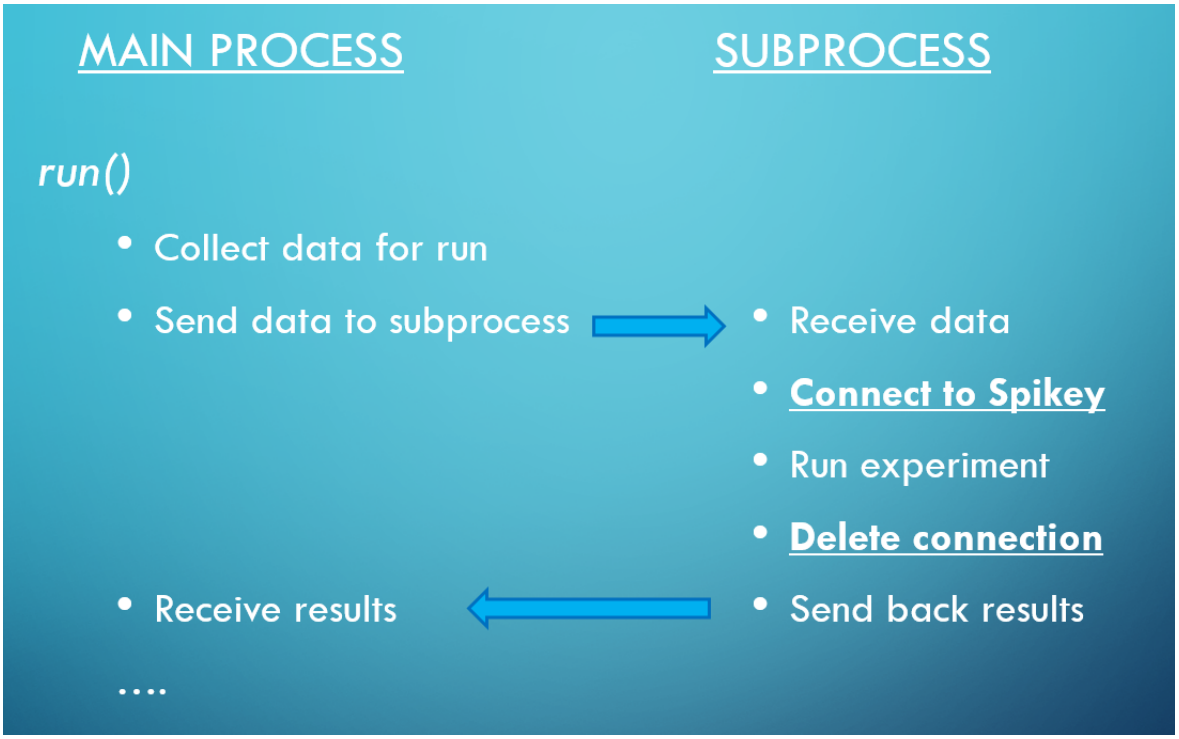
3.3 Versenden des Subprozess via Slurm

Als letzter Schritt wurde nicht mehr das gesamte Programm per Slurm gestartet, sondern nur noch der Subprozess. Das Hauptprogramm läuft komplett lokal und startet nicht mehr einfach die seperate Datei, sondern schickt sie an Slurm um auf dem Server zu laufen. Für die Kommunikation zwischen Haupt- und Unterprogramm werden nun nicht mehr Dateien verwendet in die alles gespeichert wird, sondern das Paket ZMQ. Hiermit lassen sich Nachrichten (z.B. speziell Python-Objekte die durch pickle verpackt werden können) direkt zwischen zwei Python-Prozessen versenden.
Dafür startet das Hauptprogramm in *PyNN.run()* den Subprozess mithilfe von *subprocess.Popen('srun ...')*. Mit ZMQ wird dann ein socket erstellt und "run_dataän den Subprozess geschickt. Im Subprozess wird ebenfalls ein socket erstellt, der auf eine Nachricht vom Hauptprogramm wartet. Sobald diese ankommt wird das Experiment, wie zuvor, durchgeführt. Am Ende des Subprozess wird wieder die Hardware-Verbindung gelöscht und alle Ergebnisse des Experiments per ZMQ zurück an den Hauptprozess geschickt (verpackt als dictionary "run_data_back"). Damit ist der Subprozess beendet

und es kann bereits das nächste Programm auf der Workstation laufen während unser Hauptprogramm evtl. immer noch läuft und z.B. die Daten auswertet.
Hierdurch wird der Server nur für die Hardware-Nutzung verwendet, weitere Berechnungen laufen lokal ab und verbrauchen nicht unnötig Serverkapazitäten. Der große Vorteil hiervon zeigt sich in Jupyter Notebooks, die nun parallel mit einem Chip arbeiten können. Zuvor mussten Programme zunächst in eine einzelne Datei geschrieben und diese per Slurm gestartet werden. Nun können die Notebooks lokal laufen gelassen werden, wie sie eigentlich gedacht sind. Wenn *PyNN.run()* aufgerufen wird werden die entsprechenden Daten an die Workstation geschickt, durchgeführt und der Chip/Server wieder freigegeben.
Ein Nebeneffekt dieser Implementierung ist, dass der workStationName nicht mehr aus der Umgebung gelesen werden kann wie zuvor, da das Hauptprogramm jetzt lokal läuft. Anstatt die Workstation beim Aufruf ("srun ... –gres=station x") zu spezifizieren wird der Name nun als weiterer Parameter in *PyNN.setup(workStationName=...)* festgelegt. Wird dieser Parameter nicht angegeben versucht das Programm wie zuvor den Namen in seiner Umgebung zu finden. Dies funktioniert allerdings nur wenn man in der Umgebung des Chips arbeitet (z.B. FP-Computer), ansonsten wird ein Fehler geworfen.
Falls dies der Fall ist und man doch rein lokal arbeitet wird ebenfalls kein Slurm vorhanden sein. In *PyNN.setup()* wird daher aus der Umgebung überprüft ob Slurm vorhanden ist, bzw. ob srun verwendet werden kann. Falls ja wird ein kleines Skript, get_node.py, in *PyNN.getSubprocessNode(workStationName* aufgerufen um den genauen Node herauszufinden, den der Subprozess später haben wird. Dies wird benötigt um mit ZMQ die Nachrichten zu versenden. Falls srun nicht vorhanden ist wird der Subprozess später ebenfalls lokal ausgeführt.

An dieser Stelle zeigte sich eine weitere Herausforderung: viele runs hintereinander. Viele Experimente lassen einen run auf dem Chip laufen und sind danach abgeschlossen. Wenn man aber z.B. oft denselben Versuch laufen lassen möchte und jeweils nur einen Parameter ein wenig verändert enstehen viele fast identische Prozesse hintereinander. Die Subprozesse verschiedener Nutzer kommen jedoch vermischt miteinander auf dem Server an und müssen daher davon ausgehen, dass sie den Chip jeweils komplett neu einstellen müssen. Sendet man nämlich zwei runs kurz hintereinander an den Chip kann man nicht wissen ob in der Zwischenzeit jemand anderes ebenfalls ein komplett anderes Experiment dort hat laufen lassen. Dies sorgt für unnötig lange Laufzeiten, da der Chip ständig neu konfiguriert wird obwohl dies nicht nötig wäre.
Die Lösung hierfür ist der Funktion *PyNN.run()* einen neuen Parameter "multiple_runs" zu geben der entweder True oder False ist. Mit diesem gibt man an ob nur ein einzelner run durchgeführt wird, oder mehrere hintereinander. Ist multiple_runs True löscht der Subprozess nach Durchlaufen seines runs nicht die Hardwareverbindung und endet nicht sofort. Stattdessen wird der Subprozess behalten und startet von vorne, auf die nächste Nachricht des Hauptprogramms wartend. Das geht so lange, bis er explizit in *PyNN.end()* beendet wird.
Dies bietet den Vorteil, dass der Chip nicht mehr bei jedem Durchlauf komplett neu eingestellt werden muss, sondern wie zuvor erkennt wenn kaum etwas geändert wurde. Der Nachteil davon ist, dass der Chip wieder längere Zeit von einem Nutzer blockiert

wird, bis dieser *PyNN.end()* aufruft. Daher sollte dieser Parameter nur dann gesetzt werden, wenn er wirklich nötig ist. Da Experimente die viele runs benötigen generell aber längere Zeit laufen müssen mehrere Nutzer sowieso auf Wartezeit gefasst sein wenn sie den Chip teilen, weshalb das zusätzliche Blockieren durch einen Nutzer nur einen unwesentlichen Beitrag leistet.



4 Fazit

4.1 Verbesserungen

Mit den aufgeführten Änderungen ist es nun möglich, dass mehrere Nutzer parallel an einem Spikey arbeiten können. Der Chip wird dabei nur noch dann benutzt wenn es tatsächlich notwendig ist und wird sofort wieder freigegeben sobald er nicht mehr in Verwendung ist.
Programme für den Spikey können nun lokal ausgeführt werden, anstatt per Slurm, und schicken die relevanten Dateien selbstständig an den Server wo sie ausgeführt werden. Dadurch können Jupyter Notebooks zeilenweise ausgeführt werden und müssen nicht mehr in einem Block stattfinden, der in eine Datei geschrieben und versendet wird. Dies macht dynamisches Arbeiten innerhalb des Programms möglich, ohne jedes Experiment immer als komplettes Programm laufen lassen zu müssen.

4.2 Änderungen in der Nutzung

Die Nutzung des Spikey ist zum Großteil unverändert. Statt die Workstation im Ausführungs-Befehl anzugeben wird der Name jetzt in
PyNN.setup(workStationName=... übergeben, oder vom Programm lokal gesucht.
Experimente mit vielen wiederholten runs laufen standardmäßig, sind aber deutlich langsamer. Um dies zu beschleunigen und temporär den Chip für alleinige Nutzung zu reservieren kann der Parameter *PyNN.run(multiple_runs=False* auf True gesetzt werden, was den Chip für andere Nutzer blockiert bis er durch *PyNN.end()* wieder freigegeben wird.