ARM-Cluster

Bericht für das Projekt „Parallelrechnerevaluation“

Jonas Gresens, Lennart Bergmann, Rafael Epplée

Betreuung durch Dr. Julian Kunkel & Dr. Michael Kuhn

11. Oktober 2015

Wir erklären, wie wir ein einfachen Cluster aus 5 Banana Pis gebaut haben und ob sich günstige ARM-Rechner als Clustereinheiten lohnen.

1 Projektziel

Das ursprüngliche Ziel war der Bau einen lauffähigen Cluster aus Einplatinencomputern (wie z.B. Raspberry Pis). Das ganze Projekt dient daher als Machbarkeits- bzw. Brauchbarkeitsstudie von einem kostengünstigen Cluster auf Basis der ARM-Architektur, insbesondere im Hinblick auf Stromeffizienz und Softwareunterstüzung. Ein Anwendungsszenario für einen solchen Cluster wäre beispielsweise die Ausführung von Tests im kleineren Maßstab auf echter Hardware.

Es war geplant, CoreOS als Betriebssystem und Docker zur Organisation der Anwendungen zu verwenden, da diese Kombination einen einfachen beständigen Betrieb des Clusters versprechen zu schien.

Zum Schluss sollte der Cluster, samt aller zugehöriger Hardware, in einem, für Serverschränke genormten, Plexiglas-Gehäuse Platz finden, damit er in das Cluster-Rack der Arbeitsgruppe eingebaut werden kann.

2 Hardware

2.1 Board

Als Erstes mussten wir uns Gedanken über die Wahl des Einplatinen-ARM-Computers machen. Dazu haben wir die verschiedenen Boards mit einander verglichen:

Board	Raspberry Pi 2	BeagleBone Black	Cubieboard4	Banana Pi M2
CPU	Cortex A7	Cortex A8	Cortex A7/A15	Cortex A7
Architektur	ARMv7	ARMv7a	ARMv7/ARMv7	ARMv7
Kerne	4	1	4/4	4
Taktrate	900 MHz	1000 MHz	1300/2000 MHz	1000 MHz
RAM	1 GiB DDR2	512 MiB DDR3	2 GiB DDR3	1 GiB DDR3
Netzwerk	bis 100 Mbit/s	bis 100 Mbit/s	bis 1000 Mbit/s	bis 1000 Mbit/s
Kosten	40 €	55 €	100 €	60 €

Mit vier Kernen á 2 GHz, 2 GiB DDR3 RAM und Gigabit-Ethernet ist das Cubieboard4 unschwer als das Leistungsstärkste der vier aufgeführten zu erkennen. Der Preis von 100 €ist für die Leistung zwar komplett angemessen, für unser Projekt jedoch etwas zu hoch, sodass das Cubieboard4 leider aus der engeren Auswahl fiel.

Anders beim BeagleBone Black, hier passt der Preis von nur 55,00Euro, leider kann das Board leistungstechnisch nicht mit den beiden anderen verbliebenden Boards mithalten. Mit nur einem Kern und 512 MB RAM, steht es klar hinten an.

Der Raspberry Pi 2 (Modell B) besitzt zwar einen Preisvorteil gegenüber dem Banana Pi M2, hat dafür aber auch nur DDR2 statt DDR3 RAM und eine 100 MHz niedrigere Taktfrequenz.

Ausschlaggebend war am Ende die höhere Netzwerkdurchsatzrate des Banana Pi M2, der mit 1000Mbit/s aufwarten kann. Wir haben uns für fünf Banana Pis entschieden, vier Compute-Nodes und ein Head-Node.

2.2 Weitere Komponenten

- SD-Karten

  Da die meisten nötigen Komponenten des Clusters nur auf dem Head installiert werden sollten, haben wir für diesen eine 32GB Micro-SD-Karte besorgt und für die Nodes, auf denen eigentlich nur die Berechnungen stattfinden sollen und nicht viel installiert sein muss, vier 8 GB Karten.

- Switch

  Wir haben uns für einen D-Link DGS-10008D Switch entschieden, ausschlaggebend waren hierbei das Gigabit LAN sowie die 8 Ports, so dass alle Compute-Nodes + Head-Node gleichzeitig am Switch, und dieser auch noch am Internet angeschlossen werden kann.

- Stromversorgung

  Ein Banana Pi soll bei Höchstleistung nicht mehr als 5 Watt benötigen, deshalb haben wir uns bei der Stromversorgung für einen USB-Port von Logilink entschieden, der 50 Watt Leistung bringt und sechs USB-Anschlüsse hat. Damit hatten wir über die Dauer des Projekts auch keine Schwierigkeiten, was den Energieverbrauch angeht.

- Case

  Unser Cluster hat noch kein Case in dem die ganze Hardware unterkommt, allerdings gibt es ein 3D-gedrucktes Case in dem sich die Banana Pis befinden, sodass diese nicht lose herumliegen. Das Case ist usprünglich für Raspberry Pis gedacht, die Maße des Banana Pis sollen laut Hersteller aber gleich dem der Raspberrys sein. Wie wir leider herausfinden mussten entspricht dies nicht ganz der Wahrheit, die Banana Pis sind etwas größer. Das Case musste daher von Hand angepasst werden. Die Banana Pis passen nun hinein, sitzen aber nicht so gut und fest wie es Raspberry Pis tun würden.

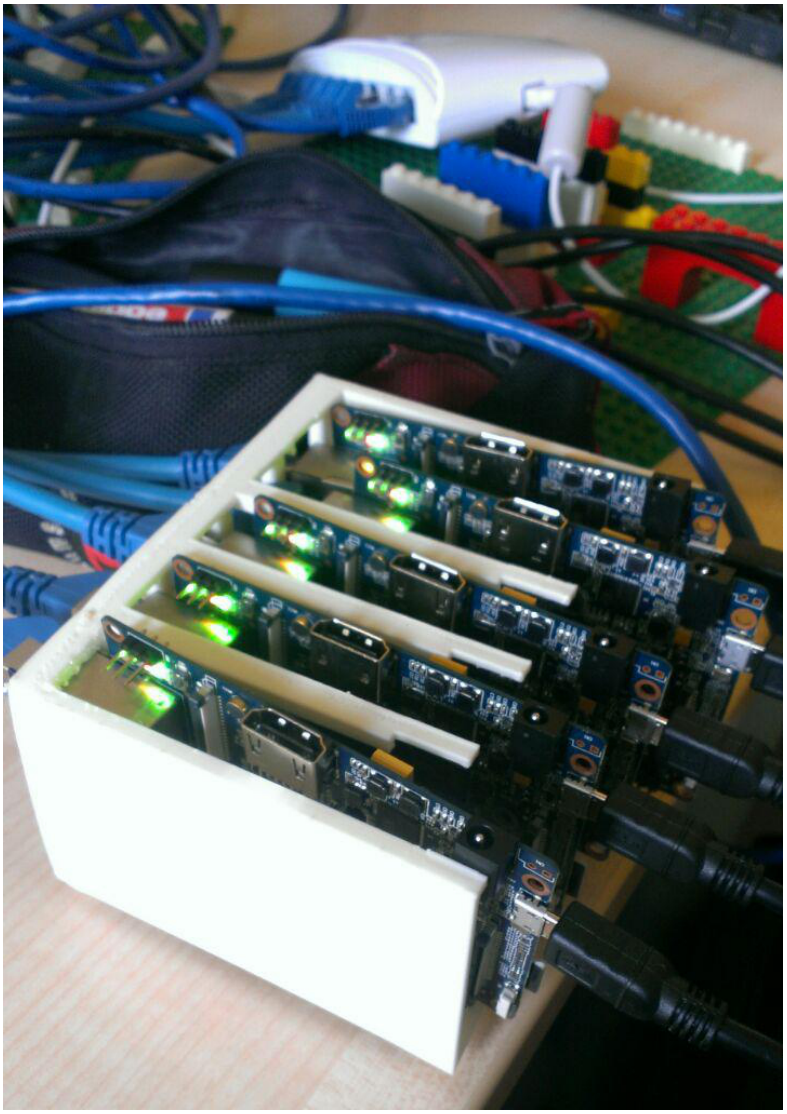

Abbildung 1: Das Herz unseres Clusters: 5 Banana Pis in einem 3D-gedruckten Case

3 System-Aufbau

3.1 OS

Eines unserer primären Ziele war es, die Anbindung neuer Compute Nodes so einfach wie möglich zu machen. Zusätzlich sollte der Cluster für eine breite Menge an Anwendungen zur Verfügung stellen. Für solche Anforderungen bietet sich CoreOS besonders an:

- Verteiltes, automatisches Konfigurationsmanagement über `etcd` (inklusive IP-Adressen-Vergabe)
- Isolierte Umgebungen mit individuellen Abhängigkeiten für jede Anwendung über Container (`rkt`)
- Anwendungsverwaltung à la Slurm über `fleet`

Leider wird die ARM-Architektur, wie wir schnell herausfanden, (noch) nicht von CoreOS unterstützt. Somit entschieden wir uns gegen CoreOS und beschlossen, die Lösung für unsere Anforderungen auf einer anderen Ebene als dem Betriebssystem zu suchen. Auf den Banana Pis läuft nun die Raspbian-Distribution, zu finden auf der offiziellen Banana-Pi-Homepage[1].

3.2 Verkabelung

Der Aufbau des Clusters ist verhältnismäßig simpel. Die 5 Banana Pis werden, sowohl Head als auch Compute Nodes, direkt mit dem Switch verbunden. Vom Switch geht ein Netzwerkkabel in das umliegende Netzwerk.

Der USB-Port liefert über 6 Ausgänge Strom, durch 5 davon werden die Banana Pis versorgt, an den 6. wird der Switch angeschlossen.

3.3 Netzwerk

Die Topologie unseres Clusters ist, wie im vorigen Abschnitt bereits zu erahnen, etwas ungewöhnlich. Bei den üblichen Clustern ist die Head Node über ein Interface an das umliegende Netzwerk und über ein anderes an den Switch und damit die restlichen Compute Nodes angebunden. Das bedeutet, dass jede Kommunikation von den Compute Nodes nach außen über die Head Node geht. Oft wird diese Topologie genutzt, um ein lokales Netzwerk zwischen den Compute Nodes aufzubauen und Kommunikation der Compute Nodes nach außen zu unterbinden. Die Head Node vergibt in diesem Fall über DHCP lokale IP-Adressen an die Compute Nodes. Diese Herangehensweise vereinfacht sowohl das Management der Compute Nodes als auch die Zugriffskontrolle auf den Cluster.

[1] `http://bananapi.com/`

Da ein Banana Pi allerdings nur einen Netzwerk-Anschluss hat, verwendet unser Cluster eine leicht abgewandelte Version dieser Topologie, wie in Abbildung 2 zu sehen. Über den Switch sind alle Nodes mit dem umliegenden Netzwerk verbunden. Statt zwei echten Netzwerk-Interfaces hat die Head Node ein virtuelles Interface, das die Verbindung zum lokalen Netzwerk darstellt. Dieses Setup funktioniert zwar, verlässt sich allerdings darauf, dass die Compute Nodes sich nicht zufällig über die direkte Verbindung an das umliegende Netz von einem anderen DHCP-Server eine IP-Adresse besorgen. Um dem Vorzubeugen, sind die Compute Nodes in unserem Fall auf der Blacklist des äußeren DHCP-Servers eingetragen und werden von ihm ignoriert. Als DHCP-Server auf der Head Node verwenden wir `dnsmasq`, in dessen Konfigurationsdateien die einzelnen Compute Nodes über die Option `dhcp-host` per MAC-Adresse mit einer festen IP assoziiert werden.

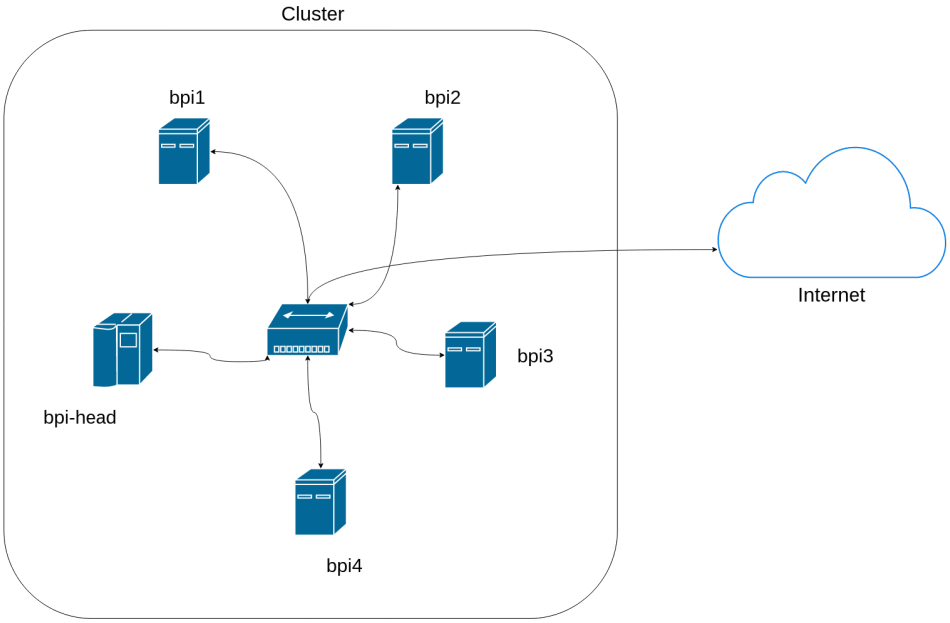

Abbildung 2: Netzwerktopologie des Clusters

3.4 NFS

Da wir fast alle Daten möglichst zentralisiert auf der Head-Node speichern, verwenden wir NFS um diese den Compute-Nodes über das Netzwerk als POSIX-kompatibles Dateisystem zugänglich zu machen:

- Auf der Head-Node läuft der NFS-Server, der `/srv` und `/home` bereitstellt.
- Auf jeder Compute-Node läuft ein NFS-Client, sodass die home-Verzeichnisse auf dem Head per fstab auf den Compute-Nodes gemountet werden können und die Compute-Nodes ihre lokalen Packages mit dem Golden Image auf dem Head synchronisieren können (siehe 3.5).

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
/srv *(rw,sync,no_subtree_check,no_root_squash)
/home *(rw,sync,no_subtree_check)
```

Listing 1: **/etc/exports** - Konfiguration des NFS-Servers

```
proc            /proc           proc    defaults          0       0
/dev/mmcblk0p1  /boot           vfat    defaults          0       2
/dev/mmcblk0p2  /               ext4    defaults,noatime  0       1

bpi-head:/home       /home       nfs
 ↪  async,rw,relatime,rsize=1048576,wsize=1048576,proto=tcp,intr,nfsvers=3       0  0
```

Listing 2: **/etc/fstab** - Konfigration des NFS-Clients

3.5 Golden Image

Cluster setzen sich aus mehreren einzelnen Rechnern zusammen und sind daher in ihrer Administration deutlich aufwändiger als ein einzelnes System. Um diese dennoch möglichst einfach zu halten, wird für alle Compute-Nodes nach Möglichkeit die gleiche Hardware verwendet und sämtliche Software der Compute-Nodes (inklusive Betriebssystem) zentral gespeichert und übers Netzwerk bereitgestellt.

Das auf der Head-Node gespeicherte Systemabbild wird als „Golden Image“ bezeichnet und stellt den gemeinsamen aktuellen Zustand aller Compute-Nodes dar. Durch das Golden Image müssen effektiv nur noch zwei verschiedene Systeme administriert werden, was die Skalierbarkeit des Cluster-Setups drastisch steigert indem es den Aufwand für die Verteilung von neuer Software und deren Konfiguration konstant hält. Die Compute-Nodes der meisten handelsüblichen Cluster booten per PXE direkt das, im Netzwerk bereitgestellte, Golden Image.

Banana Pis können jedoch nicht per PXE o.ä. über das Netzwerk gebootet werden, weil ihr BIOS so konfiguriert ist, dass es den Bootloader auf der SD-Karte benutzt, weswegen wir einen kleinen Umweg gehen mussten:

- Die SD-Karte einer jeden Compute-Node enthält ein komplett lauffähiges System, das beim Einschalten gebootet wird.
- Alle Daten auf der SD-Karte stammen aus dem Golden Image. Die Installationen unterscheiden sich nur in ihrem hostname.
- Wir verteilen Änderungen am Golden Image nicht direkt beim Reboot, sondern indem wir semiautomatisch per Skript die lokale Installation mit dem Golden Image synchronisieren.
- Die SD-Karten neuer Compute-Nodes können ebenfalls per Skript mit dem aktuellen Zustand des Golden Image geflasht werden, sodass der Cluster einfach erweiterbar bleibt.

3.5.1 Skripte

Im folgenden werden unsere drei selbstentwickelten Skripte zur Verwaltung und Nutzung des Golden Image vorgestellt:

- `create-local-installation.sh` (siehe 7.2)
  - Komplett neue Installation des aktuellen Golden Image für eine neue/kaputte Node
  - Ausführung auf einem extra Gerät (z.B. Laptop) mit SD-Karten-Slot

  - Funktionsweise: erzeugt neue Partitionstabelle, erzeugt Dateisysteme, kopiert Bootloader, kopiert Inhalt des Golden Image mit rsync über NFS von der Head-Node

- `update-local-installation.sh` (siehe 7.3)
  - Aktualisierung der lokalen Installation auf einer bereits installierten Node
  - Ausführung auf der laufenden Compute-Node
  - Funktionsweise: kopiert alle geänderten neuen Dateien mit rsync über NFS aus dem Golden Image
- `start-chroot.sh` (siehe 7.4)
  - Wrapper-Skript für die Administration des Golden Image per chroot-Environment
  - Ausführung auf der Head-Node, nur eine Instanz gleichzeitig möglich
  - Funktionsweise: mounten aller benötigten Partitionen, starten einer bash im chroot für den Nutzer

3.6 Container

Trotz unserer Entscheidung gegen CoreOS gefiel uns die Idee der Anwendungsisolierung über Container. Im Gegensatz zu virtuellen Maschinen bringen Container weniger Performanceeinbußen mit sich, was uns bei der ohnehin schon unterdurchschnittlichen Leistung der Banana Pis besonders gelegen kam.

3.6.1 Docker

Da Docker momentan die beliebteste Container-Implementation darstellt, begannen wir hier mit unseren Installationsversuchen.

Docker verwendet einige neue Features des Linux-Kernels, die in der offiziellen Raspbian-Distribution für den Banana Pi nicht enthalten sind. Die exzellente Arch-Linux-Distribution für ARM[2] unterstützt diese out of the box, da der restliche Cluster des DKRZ allerdings komplett auf Debian-basierten Systemen läuft, wurde beschlossen, für einfachere Maintenance auch auf unserem Cluster ein solches System zu verwenden. Auch auf Raspbian soll es möglich sein, Docker zum laufen zu bringen, allerdings braucht man dort einen selbst kompilierten, neueren Kernel. Im Prinzip kein Problem - leider brauchen die Banana Pis einen speziellen, eigenen Kernel vom Hersteller LeMaker[3]. Der Quelltext wurde erst vor kurzem von LeMaker bereitgestellt[4] und **ist noch bei Version 3.4, während**

[2] `http://archlinuxarm.org/`
[3] `http://www.lemaker.org/`
[4] `https://github.com/LeMaker/linux-sunxi`

Docker mindestens Version 3.10 braucht. Nach einer Menge Recherche und einem fehlgeschlagenen Versuch, einen aktuellen Linux-Kernel auf einem Banana Pi zu kompilieren, gaben wir also neben CoreOS auch Docker auf.

3.6.2 rkt

rkt ist eine alternative Container-Implementierung, entwickelt vom CoreOS-Team. rkt hat zwar keine offizielle Unterstützung für die ARM-Architektur - das hinderte uns allerdings nicht daran, das Projekt selber zu kompilieren. Dabei kam dann heraus, dass rkt leider auch (noch) keine 32-bit-Systeme unterstützt, was das Projekt auf den ARMv7-Prozessoren der Banana Pis nicht lauffähig macht.

3.6.3 systemd-nspawn

Nach unseren Versuchen mit Docker und rkt legten wir unsere letzte Hoffnung auf das systemd-initsystem. Das hat seit einiger Zeit eine minimale Container-Implementierung, genannt `systemd-nspawn`[5]. Ursprünglich für schnelle Tests von systemd selber gedacht, enthält es inzwischen die grundlegendsten Features und hätte für unsere Zwecke vollkommen gereicht. Gegen Ende unseres Projekts fingen wir damit an, systemd testweise auf einem Banana Pi zu installieren (unsere Version von Raspbian verwendete noch klassische initscripts). Die Installation endete jedoch jedes mal mit einem nicht bootenden Banana Pi, einer unbrauchbaren Raspbian-Installation und einem zeitaufwändigen neuflashen der SD-Karte. An diesem Punkt des Projekts hatten wir leider nicht mehr genug Zeit, um diesen Fehler zu beheben.

[5] `http://www.freedesktop.org/software/systemd/man/systemd-nspawn.html`

4 Installierte Software

Nachdem unseren Cluster endlich lauffähig war, wollten wir die tatsächliche Leistungsfähigkeit der Banana Pis mit der HPL-Benchmark[6] testen. HPL benötigt eine MPI- und eine BLAS-Bibliothek, die wir daher ebenfalls installieren mussten.

4.1 MPI

Der „Message Passing Interface"-Standard (MPI) beschreibt den Nachrichtenaustausch zwischen einzelnen parallel Prozessen, die gemeinsam an der Lösung eines Problems arbeiten. MPI legt dabei kein konkretes Protokoll oder Implementierung fest, sondern beschreibt die Semantik der verschiedenen Arten von Kommunikations-Operationen und ihre API, sodass die eigentliche Nutzung des Standards eine MPI-Implementierung benötigt.

Anfänglich wollten wir OpenMPI oder MVAPICH2 nutzen, die jedoch aus verschiedenen Gründen beide nicht funktionierten:

- OpenMPI ließ sich nicht vollständig kompilieren, da es auf ARM nicht lauffähigen Assembler-Code enthält und der Portierungsaufwand sich vermutlich nicht gelohnt hätte.
- MVAPICH2 funktionierte bis zu einem bestimmten Systemupdate einwandfrei, danach ließ sich jedoch die „libpmi"-Header nicht mehr finden.

Aus der Not heraus entschieden wir uns daher dazu ein schon vorkompiliertes MPICH2[7] aus den Paketquellen unserer Distribution zu benutzen, da wir an dieser Stelle davon ausgingen, dass sich die Wahl der MPI-Implementierung für unseren Cluster wenn überhaupt nur sehr gering in der gemessenen Leistung wiederspiegeln würde.

4.2 OpenBLAS

Die „Basic Linear Algebra Subprograms" (BLAS) sind eine Sammlung von Routinen für grundlegende Vektor- und Matrix-Operationen. Das von uns genutzte OpenBLAs[8] ist eine optimierte BLAS-Bibliothek und musste zur Nutzung auf den Banana Pis neu kompiliert werden. Das Kompilieren von OpenBLAS auf dem Pi hat ca. 1 Stunde gedauert. Die besten Messergebnisse haben wir mit einer Version ohne Threading (mittels `USE_THREADING=0`) erzielt.

[6] `http://www.netlib.org/benchmark/hpl/`
[7] `https://www.mpich.org/`
[8] `http://www.openblas.net/`

4.3 HPL

Die „High-Performance Linpack“-Benchmark ist ein Programm zur Messung der Fließkomma-Rechenleistung eines (verteilten) Computer-Systems. HPL misst die Leistung des Systems in GFLOPS indem es ein dichtes $n \times n$-System von linearen Gleichungen ($Ax = B$) löst und errechnet die Leistung in GFLOPS.

Wie OpenBLAS mussten wir HPL speziell für unser System kompilieren, damit es möglichst effizient an die verfügbare Hardware angepasst ist.

4.3.1 Messergebnisse

HPL hat vier verschiedene Hauptparameter:

- `N` gibt die Höhe und Breite der Matrix an, das Problem wächst quadratisch mit `N`
- `NB` ist die Nachrichtengröße bei der Kommunikation zwischen den Prozessen
- `P` und `Q` beschreiben die Aufteilung der Matrix auf das Prozessgitter

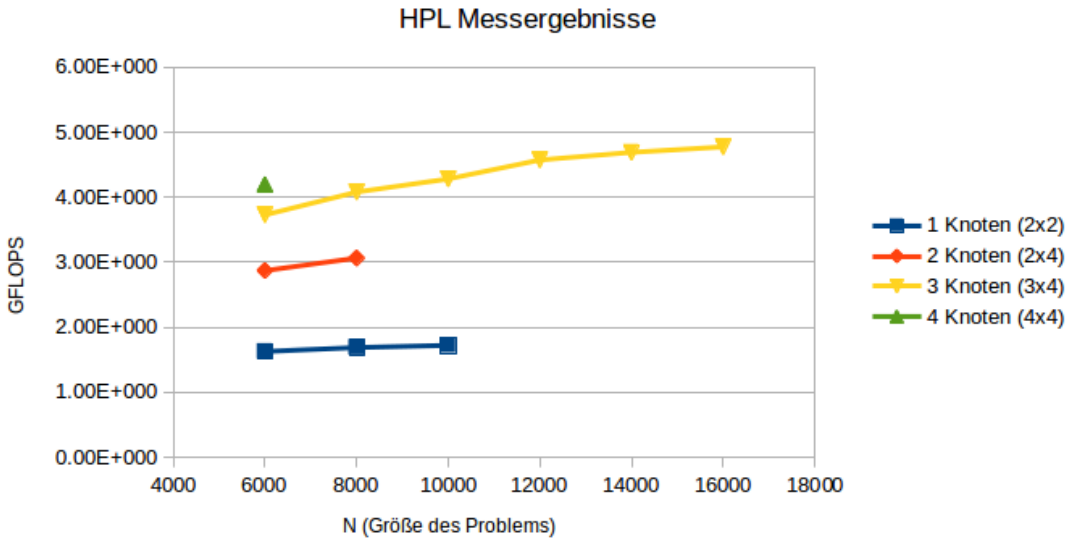

Abbildung 3: Messergebnisse mit HPL

Die vier Kurven zeigen das Verhältnis der gemessenen Performance mit verschiedenen vielen Compute-Nodes und Problemgrößen:

Der schwache Speedup (bei gleich großem Problem pro Kern) ist in Anbetracht der verwendeten Hardware überraschend gut (ungefähr 2,8 bei 3 verwendeten Compute-Nodes)

Die Entwicklung des starken Speedups (bei fester Problemgröße trotz steigender Anzahl der Kerne) zeigt jedoch, dass sich der verwendete handelsübliche Switch nicht für HPC eignet, da mit steigender Zahl an Compute-Nodes die Leistung einbricht.

Leider konnten wir keine Leistungsdaten für weitere HPL-Runs mit größerem N auf 4 Nodes messen, da es technische Probleme gab:

- `bpi4` schaltete sich bereits nach wenigen Minuten unter Volllast ab.
- `bpi-head` konnte nicht als Ersatz verwendet werden, da er die gleichen Symptome zeigte.

Die Instabilität der Pis bei großer Hitze stellt ein großes Problem für einen tatsächlich intensiv genutzten Pi-Cluster dar.

4.4 SLURM

SLURM[9] steht für Simple Linux Utility for Resource Management und ist ein Open Source Workload Manager, der auf vielen Clustern und Supercomputern rund um die Welt verwendet wird. SLURM wird dazu verwendet, einzelne Jobs möglichst effizient auf die Knoten des Clusters zu verteilen.

Obwohl wir uns sehr bemüht haben, gab es im Laufe des Projekt wesentlich mehr Komplikationen als wir erwartet haben und so fehlte uns am Ende die Zeit dafür, SLURM komplett in den Cluster zu integrieren.

Die aktuelle Konfiguration ist unter 7.5 zu finden.

[9] http://slurm.schedmd.com/

5 Status Quo

5.1 Aktueller Stand

Zum jetzigen Zeitpunkt haben wir einen zwar nicht optimalen aber funktionierenden Cluster. Wir benutzen ein Golden Image, mit dessen Hilfe es ein leichtes ist, alle Knoten auf den selben Stand zu bringen, und durch welches der Cluster einfach um zusätzliche Knoten erweiterbar ist. Die Netzwerkeinbindung ist noch suboptimal, da Compute-Nodes in manchen fällen eine IP aus dem Internet, statt von der Head-Node beziehen, wo durch sie von dieser nicht ansprechbar sind. Auf der positiven Seite, kann der Cluster mit nutzbaren MPI-Libraries, sowie lauffähigem HPL aufwarten, womit ausführliche Leistungstests möglich sind. Die Banana Pis befinden sich in einem eigens dafür 3D-gedrucktem Case, welches leider ein bisschen klein geraten ist.

5.2 Weiterführende Arbeit

Die Lösung Netzwerk-Problematik wären statische Ips, sodass keine Knoten mehr eigenwillig auf IP-Suche gehen können.

Für eine möglichst vollständige Integration in den WR-Cluster müssten nur noch ein paar wenige Softwarepakete hinzugefügt werden:

- mit SLURM bleibt das (vom WR-Cluster) gewohnte UI zur Nutzung des Clusters erhalten.
- auf dem WR-Cluster wird LDAP zum Zugriff auf die Home-Verzeichnisse der Nutzer verwendet und wird daher zwingend auf dem Pi-Cluster benötigt um die Problematik mehrerer Home-Verzeichnisse pro Nutzer zu umgehen.
- das Monitoring-Tool Ganglia ist zwar im allgemeinen optional, sollte aber auch vorhanden sein, da es wie SLURM ebenfalls zur gewohnten UI gehört.

Außerdem fehlt für den Einbau in diesen Cluster ein Plexiglas-Gehäuse in dem die ARM-Computer sowie das gesamte Zubehör, einige LEDs und Lüfter Platz finden.

6 Fazit

6.1 Der Bananana Pi

Nach einiger Erfahrung mit dem Modell „Banana Pi“ möchten wir hier einen kurzen Überblick über Vor- und Nachteile des Modells geben.

Da wir aus der Perspektive des HPC an dieses Projekt gegangen sind, waren für uns die Hardware-Aspekte bei der Planung besonders wichtig. Im Besonderen versprach der Banana Pi durch das Gigabit-Ethernet eine problemfreie Kommunikation zwischen den Compute Nodes, auch mit den für HPC üblichen großen Datenmengen. Tatsächlich konnten wir in unseren Benchmarks auch keine Probleme mit der Kommunikationslast erkennen. Zusätzlich ergaben unsere Vergleiche mit einem Raspberry Pi 2 Modell B gerade mal einen Performanceunterschied von etwa 300 MFLOPS.

Schnell mussten wir jedoch feststellen, dass die eher kleine Community und das Ökosystem um den Banana Pi einige Probleme mit sich bringen. Zum einen mussten wir feststellen, dass es tatsächlich verschiedene Hersteller gibt, die alle von sich behaupten, den „offiziellen“ Banana Pi zu fabrizieren. Mit jedem dieser Hersteller kam eine leicht andere Dokumentation mit vielen fehlenden Stellen. SD-Karten-Images für den Raspberry Pi funktionieren mit dem Banana Pi nicht; Es bleibt unklar, ob es eine tatsächlich offizielle Seite mit offiziellen Linux-Distributionen gibt. Lange Zeit war der modifizierte Linux-Kernel für den Banana Pi nicht quelloffen; Nach viel Druck aus der Community wurde der Code dann doch veröffentlicht[10]. Leider ist das letzte stabile Release dieses Kernels noch bei Version 3.4. Das machte unter Anderem das kompilieren eines eigenen Kernels für die Docker-Installation unmöglich.

6.2 Andere Vor- und Nachteile

Eine der Fragen, die wir mit unserem Projekt beantworten wollten, war die Frage nach der Stromeffizienz der Banana Pis, insbesondere im Kontext des High Performance Computing. Leider enttäuscht der ARM-Cluster in dieser Hinsicht. Wenn wir eher konservativ[11] von einem Verbrauch von 5 Watt (unter voller Last) ausgehen, entspräche das bei unseren Benchmarks ungefähr $\frac{1}{4}$ GFLOPS pro Watt. Laut der TOP500-Liste sind Werte im Bereich von 2 GFLOPS pro Watt der aktuelle Stand der Technik[12].

Trotz des geringen Preis eignet sich ein Cluster aus Banana Pis nichtmal gut für das Testen von Programmen in kleinem Maßstab, da die ARM-CPU nicht alle Befehlssätze einer x86(_64)-CPU unterstützt und daher für Probleme bei der Kompilierung sorgt - die Nutzung eines virtualisierten Clusters (z.B. via Vagrant) ist deutlich komfortabler.

[10] `https://github.com/LeMaker/linux-sunxi`
[11] `http://raspberrypi.stackexchange.com/questions/5033/how-much-energy-does-the-raspberry-pi-consume-in-a-da`
[12] `https://en.wikipedia.org/wiki/Performance_per_watt`

Im Allgemeinen kann man also sagen, dass sich ein ARM-Cluster unserer Erfahrung nach hauptsächlich lohnt, um in einem möglichst realistischen Szenario Aufbau und Installation eines Clusters zu üben.

Die Skripte zur Verwaltung der verschiedenen Nodes funktionieren exzellent und können ohne Probleme in jedem Projekt, das mit mehreren Pi-artigen Computern arbeitet, zur einfachen Administration verwendet werden.

7 Appendix

7.1 Messergebnisse

T/V	N	NB	P	Q	Time	Gflops
WR11C2R4	6000	192	2	2	88.52	1.627e+00
WR11C2R4	8000	192	2	2	202.29	1.688e+00
WR11C2R4	10000	192	2	2	387.15	1.722e+00
WR11C2R4	10500	192	2	2	448.11	1.723e+00
WR11C2R4	6000	192	2	4	50.17	2.872e+00
WR11C2R4	8000	192	2	4	111.44	3.064e+00
WR11C2R4	6000	192	4	4	34.33	4.196e+00
WR11C2R4	8000	192	3	4	83.69	4.080e+00
WR11C2R4	6000	192	3	4	38.63	3.729e+00
WR11C2R4	10000	192	3	4	155.76	4.281e+00
WR11C2R4	12000	192	3	4	251.99	4.572e+00
WR11C2R4	14000	192	3	4	390.12	4.690e+00
WR11C2R4	16000	192	3	4	572.15	4.773e+00
WR11C2R4	6000	192	4	4	34.65	4.157e+00
WR11C2R4	2000	192	4	4	3.04	1.759e+00

7.2 create-local-installation.sh

```
#!/bin/bash

# JG/RE/LB 2015

if [[ $# -ne 4 ]]; then
        echo "Usage: $0 bpi_head_ip dist device hostname"
        exit 1
fi

if [ "$USER" != "root" ]; then
        echo "MUST be run as root! Try 'sudo $0 ...'"
    exit 1
fi

BPI_HEAD_IP="$1"
DIST="$2"
DEV="$3"
HOSTNAME="$4"

# sd card formatting
echo "Partitioning $DEV"
parted -s $DEV mklabel msdos || exit 1
parted -s $DEV unit s mkpart primary fat32 8192s 122879s || exit 1
parted -s -- $DEV unit s mkpart primary 122880s -1s || exit 1
echo "Report:"
fdisk -l $DEV

echo "Filesystems:"
mkfs.vfat -F 32 ${DEV}p1 || exit 1
mkfs.ext4 ${DEV}p2 || exit 1

echo "Bootloader"
dd if="$(dirname $0)"/bootloader_ohne_table.img count=2048 of=/dev/mmcblk0 seek=8
 ↪  bs=1024

# mounting
mkdir /tmp/target/{b,r}oot -p
mount ${DEV}p1 /tmp/target/boot
mount ${DEV}p2 /tmp/target/root

# rsync magic
rsync -axzh --stats root@$BPI_HEAD_IP:/srv/nodes/$DIST/boot/ /tmp/target/boot || exit 1
rsync -axzh --stats root@$BPI_HEAD_IP:/srv/nodes/$DIST/root/ /tmp/target/root || exit 1

echo $HOSTNAME > /tmp/target/root/etc/hostname

# Clean up
umount /tmp/target/*
```

7.3 update-local-installation.sh

```
#!/bin/bash

# JG/RE/LB 2015

test -z "$1" && echo "Please specify a distribution." && exit 1

DIST="$1"

#####
# mounting the source, this could be done outside of this script to use arbitrary sources.
mkdir -p /tmp/source_root
umount /tmp/source_root
mount -o ro,nfsvers=3 bpi-head:/srv/nodes/$DIST/root /tmp/source_root || exit 1

mkdir -p /tmp/source_boot
umount /tmp/source_boot
mount -o ro,nfsvers=3 bpi-head:/srv/nodes/$DIST/boot /tmp/source_boot || exit 1

####
# Sanity checks to check if the ip is 10.0.0.X where X below 100, then the script is allowed to run.

IP=$(ip addr show dev eth0 |grep "inet " |sed "s/.* inet \([0-9.]*\)\/.*/\1/")

if [[ ${#IP} -lt 8 ]] ; then
        echo "Could not determine IP!"
        exit 1
fi

LAST=${IP#10.0.0.}
if [[ $IP == $LAST || $LAST -gt 99 ]] ; then
        echo "Invalid host with IP: $IP"
        echo "I won't run the script on this machine!"
        exit 1
fi

rsync -axzh --delete --exclude="/home" --ignore-errors --progress /tmp/source_root/ /
rsync -axzh --delete --exclude="/home" --ignore-errors --progress /tmp/source_boot/ /boot

hostname > /etc/hostname

# cleanup and restoring the previous state
umount /tmp/source_root
umount /tmp/source_boot
```

7.4 start-chroot.sh

```
#!/bin/bash

# JG/RE/LB 2015
(
flock -n 200

CHROOTDIR="/srv/nodes/default/root"

usage ()
{
        echo "USAGE: ${0} [-h|--help] [<DIR>]"
        echo
        echo "     -h"
        echo "     --help           print this message"
        echo "     <DIR>  directory with common file system"
        echo
}

if [[ "$1" != "" ]] ; then
        CHROOTDIR="/srv/nodes/${1}/root"
fi


[ -d "${CHROOTDIR}" ] || \
        { echo "Directory for chroot ${CHROOTDIR} not found!" && exit 1; }

echo "Starting chroot environment in ${CHROOTDIR}"

# mount dev
[ -d ${CHROOTDIR}/dev ] &&
mount -o bind /dev ${CHROOTDIR}/dev

# mount dev/pts
[ -d ${CHROOTDIR}/dev/pts ] &&
mount -o bind /dev/pts ${CHROOTDIR}/dev/pts

# mount /run for resolv.conf
[ -d ${CHROOTDIR}/run ] &&
mount -o bind /run ${CHROOTDIR}/run

# mount boot 'partition'
mount -o bind ${CHROOTDIR}/../boot ${CHROOTDIR}/boot

# mount proc
[ -d ${CHROOTDIR}/proc ] &&
mount -t proc proc_chroot ${CHROOTDIR}/proc

# mount sysfs
[ -d ${CHROOTDIR}/sys ] &&
mount -t sysfs sysfs_chroot ${CHROOTDIR}/sys

```

```
51
52  #JK:
53  #sed -i "s/10.0.0.250/129.206.100.126/" £{CHROOTDIR}/etc/resolv.conf
54  if [ -f ${CHROOTDIR}/usr/sbin/invoke-rc.d ]
55  then
56          echo '#!/bin/sh' > ${CHROOTDIR}/usr/sbin/policy-rc.d
57          echo 'exit 101' >> ${CHROOTDIR}/usr/sbin/policy-rc.d
58
59          chmod +x ${CHROOTDIR}/usr/sbin/policy-rc.d
60  fi
61
62  /usr/sbin/chroot ${CHROOTDIR} /bin/bash
63
64
65  # YOU ARE IN CHROOT HERE AND THIS SCRIPT IS STOPPED UNTIL YOU QUIT BASH!
66
67
68  echo "Closing chroot environment!"
69
70  [ -f ${CHROOTDIR}/usr/sbin/policy-rc.d ] &&
71  rm ${CHROOTDIR}/usr/sbin/policy-rc.d
72
73  # umount sysfs
74  mountpoint -q ${CHROOTDIR}/sys && umount sysfs_chroot
75
76  # umount proc
77  mountpoint -q ${CHROOTDIR}/proc && umount proc_chroot
78
79  # umount boot 'partition'
80  # `mountpoint` doesn't work here for some reason
81  umount ${CHROOTDIR}/boot
82
83  # umount dev/pts
84  mountpoint -q ${CHROOTDIR}/dev/pts && umount ${CHROOTDIR}/dev/pts
85
86  # umount /run
87  mountpoint -q ${CHROOTDIR}/run && umount ${CHROOTDIR}/run
88
89  # umount dev/pts
90  mountpoint -q ${CHROOTDIR}/dev && umount ${CHROOTDIR}/dev
91  ) 200>/var/lock/start-chroot.sh
```

7.5 SLURM Konfiguration

```
ControlMachine=bpi-head
ControlAddr=10.0.0.250
#
MailProg=/usr/bin/mail
MpiDefault=none
#MpiParams=ports=#-#
ProctrackType=proctrack/cgroup
ReturnToService=1
SlurmctldPidFile=/var/run/slurmctld.pid
#SlurmctldPort=6817
SlurmdPidFile=/var/run/slurmd.pid
#SlurmdPort=6818
SlurmdSpoolDir=/var/spool/slurmd
SlurmUser=slurm
#SlurmdUser=root
StateSaveLocation=/var/spool/slurm
SwitchType=switch/none
TaskPlugin=task/none
#
# TIMERS
#KillWait=30
#MinJobAge=300
#SlurmctldTimeout=120
#SlurmdTimeout=300
#
# SCHEDULING
FastSchedule=1
SchedulerType=sched/backfill
#SchedulerPort=7321
SelectType=select/linear
#
# LOGGING AND ACCOUNTING
AccountingStorageType=accounting_storage/none
ClusterName=pi-cluster
#JobAcctGatherFrequency=30
JobAcctGatherType=jobacct_gather/none
#SlurmctldDebug=3
SlurmctldLogFile=/etc/slurm-llnl/slurmctldLog.log
#SlurmdDebug=3
SlurmdLogFile=/etc/slurm-llnl/slurmdLog.log
#
# COMPUTE NODES
NodeName=bpi[1-4] CPUs=4 State=UNKNOWN
PartitionName=debug Nodes=bpi[1-4] Default=YES MaxTime=INFINITE State=UP
```