Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung zum Proseminar
"Python im Hochleistungsrechnen"

Überblick über die Grundfunktionen des Spack-Paketmanagers

vorgelegt von

Hauke Sommerfeld

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Software-System-Entwicklung
Matrikelnummer: 7027846

Betreuer: Dr. Michael Kuhn

Hamburg, 30.09.2019

Vorwort

Spack ist ein von Todd Gamblin[1] initiierter Paketmanager für Supercomputer, der in der Programmiersprache Python entwickelt wird und für die Benutzung auf Linux oder Unix-artigen Betriebssystemen vorgesehen ist. Er vereint die Eigenschaften von herkömmlichen Lösungen mit neuen, innovativen Ansätzen speziell im Bereich des Hochleistungsrechnens. Dabei erfreut sich Spack zunehmend immer größerer Beliebtheit. Zu den Anwendern gehören namhafte Organisationen wie das *Exascale Computing Project*[2].

Diese Ausarbeitung soll in Anlehnung an meinen Vortrag noch einmal einen Überblick über die grundlegenden Funktionen von Spack bieten. Der Inhalt orientiert sich hierbei vor allem an der online-Dokumentation von Spack[3].

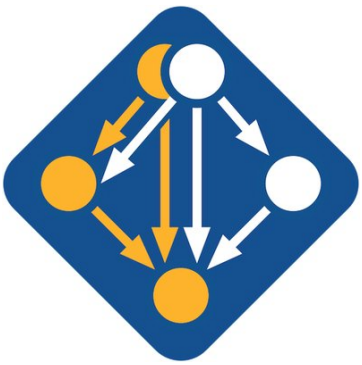

Abbildung 0.1: Das Logo des Spack-Paketmanagers

[1] https://people.llnl.gov/gamblin2
[2] https://www.exascaleproject.org/spack-the-deployment-tool-for-ecps-software-stack/ [Pro18]
[3] https://spack.readthedocs.io/en/latest/index.html [TG19]

Inhaltsverzeichnis

1 Grundlegendes

1.1 Herkömmliche Paketmanager

Software im Umfeld des wissenschaftlichen Rechnens stellt oft besondere Ansprüche an ihre Ausführungsumgebung und ist nicht selten, anders als Programme für den privaten oder Bürogebrauch, hochgradig komplex. Sie sind oftmals eng verzahnt mit dem jeweiligen System und können in Kombination mit einer Vielzahl von den im Hochleistungsrechnen üblichen Workload-Managern, MPI-Implementationen und Berechnungsbibliotheken auftreten, was das Bereitstellen einer allgemein lauffähigen Konfiguration in Form eines Binaries erheblich erschwert. Dies ist nicht zuletzt auch deswegen der Fall, da eine Konfiguration aus Performancegründen häufig bereits im Build-Prozess, bspw. durch das Linken der benötigten Bibliotheken, geschieht.

Herkömmliche Paketmanager wie `apt-get, yum` oder `dnf` stellen dem Benutzer bereits kompilierte Binaries für die jeweils benötigte Architektur bereit, welche zusammen mit den nötigen Konfigurationsskripten in Form eines Archivs, einem sogenannten Paket, aus einem Repository bezogen werden. Eine solche Funktionsweise ist für den alltäglichen Gebrauch sehr von Vorteil, da auf die teils langen Zeiten zum Kompilieren verzichtet werden kann; generell leidet darunter jedoch die auf Hochleistungrechnern notwendige Flexibilität.

1.2 Unterschiede

In einigen Bereichen verfolgt Spack andere Ansätze als herkömmliche Paketmanager, um auf die besonderen Gegebenheiten im Hochleistungsrechnen einzugehen. Dazu zählen insbesondere die folgenden Bereiche:

Packages Packages werden in Spack als reine Python-Skripte realisiert, welche alle nötigen Informationen zum Installieren des Paketes enthalten. Anders als bei herkömmlichen Paketmanagern wird die zu installierende Software auf dem System kompiliert und nicht als fertiges Binary heruntergeladen.

Konfiguration Ein solches oben beschriebenes Spack-Package definiert jeweils eine Standardkonfiguration, welche bei Bedarf vor der Installation an die eigenen Bedürfnisse angepasst werden kann. Hierzu gibt es verschiedene Möglichkeiten wie das permanente Anpassen eines Paketes an die eigenen Bedürfnisse oder das Überschreiben von einzelnen

Parametern über die Kommandozeile, auf welche ich im späteren Verlauf noch eingegangen wird.

Koexistenz Ein weiterer bedeutender Vorteil von Spack ist die Möglichkeit, mehrere Konfigurationen eines Packages parallel zu installieren, indem diese in eigenständigen Verzeichnissen installiert werden und, bspw. per `module load`, in das Environment geladen werden. Dieses Prinzip ähnelt gewissermaßen den Virtual-Environments in Python und ermöglicht es, die gleiche Software mit unterschiedlichen Compilern oder Versionen zu benutzen. Dies ist im Hochleistungsrechnen sehr von Vorteil, da wissenschaftliche Anwendungen unter den Versionen häufig inkompatibel sind oder zumindest größere Unterschiede aufweisen, sodass eine Umstellung von bestehenden Projekten nicht selten einen erheblichen Mehraufwand darstellt.

Performance Insgesamt lässt sich mit Spack durch das Automatisieren von Build-Prozessen und die Minimierung von Paket-Konflikten durch Koexistenz ein signifikanter Performancegewinn gegenüber dem manuellen Einrichten einer Softwareumgebung erzielen.
Nach Aussage Todd Gamblins, dem führenden Entwickler von Spack, ließ sich die Build-Zeit des Software-Stacks des Supercomputers *Summit* der *Oak Ridge Leadership Computing Facility (OLCF)* demnach von 2 Wochen auf etwa 12 Stunden verkürzen[1].

[1] https://twitter.com/tgamblin/status/1085938737924788229

2 Funktionsweise von Spack

2.1 Installation

Spack wird auf GitHub entwickelt und der Inhalt des Repositories[1] stellt zu jedem Zeitpunkt auch den minimalen Inhalt der lokalen Spack-Installation dar. Das Installieren des Paketmanagers erfolgt dementsprechend rein über *Git*, genauer über den Aufruf von `git clone` im gewünschten Installationsverzeichnis. Die Bedienung von Spack erfolgt Unix-typisch über eine Reihe von Shell-Befehlen, auf die im nächsten Unterkapitel genauer eingegangen wird. Zur einfacheren Bedienung kann Spack durch eine Reihe von mitgelieferten Skripten zudem ins jeweils aktuelle Environment eingebunden werden.

2.1.1 Beispiel

Das Einrichten einer minimalen Spack-Installation kann dann bspw. wie folgt aussehen:

```
1 git clone https://github.com/spack/spack.git
2 source spack/share/spack/setup-env.sh
```

Listing 2.1: Minimaler Ablauf einer Installation

2.1.2 Überblick über die Ordnerstruktur

Die Ordnerstruktur im Spack-Installationsverzeichnis ist der eines POSIX-Systems sehr ähnlich, wie an der nachfolgenden Tabelle leicht ersichtlich ist:

`bin`	Spack-Binaries
`etc`	Konfigurationsdateien, etc.
`lib`	Benötigte Bibliotheken, Dokumentation, etc.
`opt`	Installationsverzeichnis für Pakete
`share`	Hilfreiche Skripte und Utilities
`var`	Build- und Package-Dateien

Einen besonder Fokus soll hier auf den Verzeichnissen `etc` und `var` liegen, da diese in direktem Zusammenhang mit der Konfiguration von Spack stehen, bzw. letzteres einen guten Einblick in die technische Umsetzung der Koexistenz von Paketen bietet.

[1] https://github.com/spack/spack

An dieser Stelle sei ebenfalls erwähnt, dass zwar der Eindruck entsteht, man könne die Installation durch geschicktes Symlinken tiefer in das jeweilige System integrieren, dies allgemein aber als schlechte Praxis angesehen und seitens der Entwickler eine Verwendung von Environment-Modules vorsehen wird[2].

2.2 Einführung in die grundlegende Verwendung

Im Folgenden soll eine Übersicht über die grundlegenden Spack-Befehle gegeben werden.

2.2.1 Wichtige Befehle

Anzeigen aller verfügbaren Pakete

Alle zur Installation verfügbaren Softwarepakete lassen sich durch den Befehl `spack list` auflisten. Spack reagiert sann mit einer Auflistung ähnlich jener in Listing 2.2.

```
1 -bash-4.2$ spack list
2 ==> 3121 packages.
3 abinit            multitail       r-blob
4 abyss             multitime       r-blockmodeling
5 accfft            multiverso      r-bookdown
6 ack               mummer          r-boot
7 [...]             [...]           [...]
```

Listing 2.2: Auflistung der verfügbaren Pakete

Informationen zu einem Paket

Weitere Informationen zu einem jeweiligen Paket lassen sich in Spack über den Befehl `spack info <package>` erfragen, wobei der Parameter durch den zuvor aufgelisteten Paketnamen zu ersetzen ist.

Wie im Beispiel 2.3 erkennbar ist, besteht die Ausgabe unter anderem aus den folgenden häufig genutzten Kerninformationen: Einer kurzen und allgemeinen Beschreibung, dem URL zu einer Projekthomepage, seinen assoziierten Tags (intern zur Kategorisierung verwendet) sowie den verfügbaren Versionen.

```
1 -bash-4.2$ spack info zlib
2 Package:   zlib
3
4 Description:
```

[2] "The use of module systems to manage user environment in a controlled way is a common practice at HPC centers [...]"[TG19]

```
5       A free, general-purpose, legally unencumbered
6       lossless data-compression library.
7
8   Homepage: http://zlib.net
9
10  Tags:
11      None
12
13  Preferred version:
14      1.2.11      http://zlib.net/fossils/zlib-1.2.11.tgz
```

Listing 2.3: Informationen zu einem Paket

Abhängigkeiten eines Paketes

die meiste Software benötigt wiederum andere Software wie Bibliotheken, um lauffähig zu sein. Wie in anderen gängigen Paketmanagern auch, können Pakete in Spack hierfür Abhängigkeiten spezifizieren, welche dann bei der Installation des Paketes mit installiert werden, sofern diese auf dem System noch nicht (in der korrekten Version) vorliegen.

Zum Auflisten der Abhängigkeiten bietet Spack mit dem Befehl `spack spec <package>` eine gut strukturierte Lösung an, welche die Abhängigkeiten und deren genaue benötigte Konfiguration baumartig auf dem Terminal ausgibt. Die Konfiguration umfasst neben der Version noch weitere Aspekte wie den Compiler, auf welche im späteren Verlauf eingegangen wird.

```
1   -bash-4.2$ spack spec gdbm
2   Input spec
3   --------------------------------
4   gdbm
5
6   Concretized
7   --------------------------------
8   gdbm@1.18.1%gcc@4.8.5 arch=linux-centos7-x86_64
9       ^readline@7.0%gcc@4.8.5 arch=linux-centos7-x86_64
10          ^ncurses@6.1%gcc@4.8.5~symlinks~termlib [...]
11              ^pkgconf@1.5.4%gcc@4.8.5 arch=linux-[...]
```

Listing 2.4: Auflisten von Paketabhängigkeiten

Installieren eines Paketes

Pakete können wit Hilfe des Befehls `spack install <package>` installiert werden. Die benötigten Abhängigkeiten werden hierbei automatisch mitinstalliert.

```
-bash-4.2$ spack install zlib
==> Installing zlib
==> Searching for binary cache of zlib
==> No binary for zlib found: installing from source
  ↪ /zlib-1.2.11.tar.gz
==> Staging archive:
  ↪ [...]/var/spack/stage/zlib-1.2.11-<hash>.tar.gz
==> Created stage in
  ↪ [...]/var/spack/stage/zlib-1.2.11-<hash>
==> No patches needed for zlib
==> Building zlib [Package]
==> Executing phase: 'install'
==> Successfully installed zlib
  Fetch: 0.03s.  Build: 3.23s.  Total: 3.26s.
[+] [...]/spack/opt/spack/<os>/<compiler>/zlib-[...]
```

Listing 2.5: Installieren eines Pakets

Legende: [...] = Auslassung; <...> = Element

Auflisten aller installierten Pakete

Das Auflisten der auf dem System installierten Pakete erfolgt über den Befehl `spack find`. Die Auflistung erfolgt hier geordnet nach bspw. dem verwendeten Compiler, sonst aber ähnlich wie beim Anzeigen der verfügbaren Pakete:

```
-bash-4.2$ spack find
==> 4 installed packages
-- linux-centos7-x86_64 / gcc@4.8.5 ------------------
libsigsegv@2.11  m4@1.4.18  zlib@1.2.11

-- linux-centos7-x86_64 / ncc@2.1.0 ------------------
zlib@1.2.11
```

Listing 2.6: Auflisten aller installierten Pakete

Deinstallieren eines Paketes

Mit dem Befehl `spack uninstall <package>` können zuvor installierte Pakete deinstalliert werden. Liegt ein Paket in mehreren Konfigurationen vor, ist es unter Umständen nötig, die Auswahl durch weitere Parameter einzugrenzen. Ungenutzte Abhängigkeiten werden standardmäßig ebenfalls vom System entfernt (siehe Listing 2.7).

```
-bash-4.2$ spack uninstall m4
==> The following packages will be uninstalled:

        -- linux-centos7-x86_64 / gcc@4.8.5 ----------
        hhsvnbb m4@1.4.18%gcc patches=[...]
==> Do you want to proceed? [y/N]
```

Listing 2.7: Deinstallieren eines Paketes

2.2.2 Paketkonfigurationen

Wie bereits am Anfang erwähnt, ist es häufig nötig, die genaue Konfiguration eines Spack-Paketes auf der Kommandozeile angeben zu können, um bspw. bei der Installation explizit eine andere als die Standardkonfiguration eines Paketes auszuwählen.
Spack bietet zu diesem Zweck eine Syntax bestehend aus einer Reihe von Suffixen an, welche einem Paketnamen angehangen werden können, um seine genaue Konfiguration anzugeben. Diese setzt sich im Kern aus folgenden Elementen zusammen:

- `@` Angabe einer Versionsnummer
- `%` Verwendung eines bestimmten Compilers
- `^` Hinzufügen einer Paketabhängigkeit
- `+` Setzten Boolean Compile-Time Konstante (Flag)
- `=` Setzen einer sonstigen Konstanten

Auf alle der oben aufgelisteten Modifikatoren folgt, ähnlich einem Parameter, ein Wert, auf den die Eigenschaft geändert werden soll. Die Modifikatoren können beliebig miteinander kombiniert werden und sind nicht persistent, was bedeutet, dass die Dateien ein Packages bearbeitet werden müssen, um bspw. eine weitere Abhängigkeit persistent zu machen.

```
glib@2.60.1
zlib%gcc+debug
openfoam@1812\%ncc^openblas
```

Listing 2.8: Beispiel für Paketmodifikatoren

2.3 Packages in Spack

Spack-Packages sind Python-Klassen, welche Metadaten, eine URL zum Sourcecode der Software sowie die notwendigen Installationsskripte enthalten. Da Programme bei Spack vorzugsweise aus ihrem Quellcode installiert werden liegt kein Binary bei.

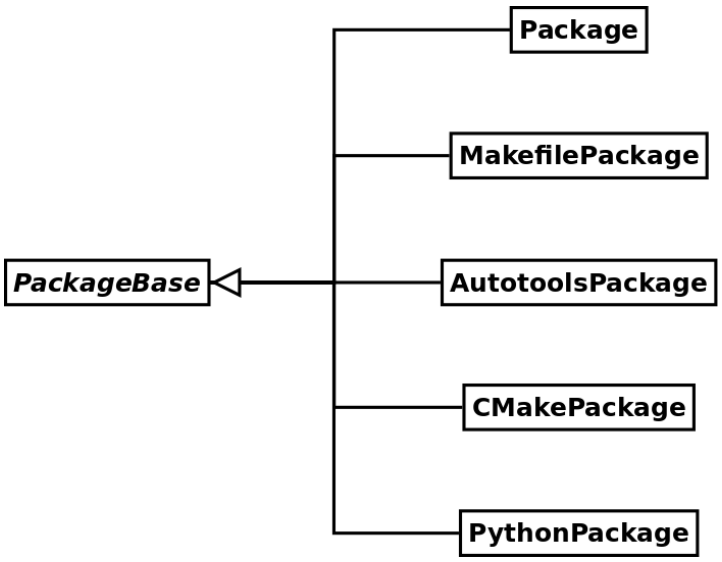

Abbildung 2.1: Subklassen von `Package`

Alle Packages erben bei Spack indirekt von der Klasse `PackageBase`. Für konkrete Implementationen gibt es eine Reihe von Subklassen, welche in Abbildung 2.1 dargestellt sind und Standardimplementationen für gängige Anwendungsfälle bieten. Die Optionen reichen hier von der simplen Klasse `Package` mit wenig eigener Funktionalität, über die Klasse `MakefilePackage` für Makefile-Projekte bis hin zu komplexeren Implementationen. Es ist ebenfalls möglich, eigene Klassen zu schreiben.

Repositories

Spack Packages werden in Repositories verwaltet, wobei dies auf zwei verschiedenen Wegen geschieht:

1. Spack liefert derzeit rund 3500[3] Pakete mit, deren Python-Dateien sich im Verzeichnis `var/spack/repos/builtin/` befinden. Über GitHub werden hier von den Entwicklern regelmäßig neue Pakete eingepflegt.

2. Alternativ besteht seitens des Nutzers die Möglichkeit, eigene Repositories separat zu pflegen und diese in Spack einzubinden. Dies geschieht über die YAML-Konfigurationsdatei `~/.spack/repos.yaml`, in die der Pfad zum Repository eingetragen und fortan von Spack berücksichtigt wird.

Beide Arten von Repositories sind strukturell gleich aufgebaut und bestehen aus einer `packages.yaml`-Datei, welche Metadaten zu den einzelnen Paketen enthält sowie den Python-Dateien mit den Paketklassen selber. Aus Gründen der Übersichtlichkeit möchte ich im Folgenden allerdings nur auf die grundlegenden Aspekte von Packages eingehen.

[3]Stand September 2019, https://spack.readthedocs.io/en/latest/package_list.html

Interner Aufbau von Spack-Packages

```
1  from spack import *
2
3  class Helloworld(Package):
4      """Package description"""
5
6      homepage = "http://www.example.com"
7      url      = "helloworld"
8
9      version('1.2.3', '0123456789abcdef01234[...]')
10
11     depends_on('foo')
12
13     def install(self, spec, prefix):
14         make()
15         make('install')
```

Listing 2.9: Beispiel-Package

Listing 2.9 zeigt den Inhalt der `package.py`-Datei eines Beispiel-Packages. Wir können ein solches Package komfortabel vom Terminal aus mit dem Befehl `spack create <name>` erzeugen, was den Vorteil hat, dass es gleich in das Spack-Setup integriert ist und sein Grundgerüst nicht von Hand geschrieben werden muss. Standardmäßig wird es dann mit dem in der Umgebungsvariablen `$EDITOR` hinterlegten Code-Editor zum Bearbeiten geöffnet.

Metadaten Direkt nach dem Kopf der Klasse in Zeile 3 folgt ein Docstring, welcher als Beschreibung des Packages dient. In den darauf folgen Zeilen finden sich Felder mit den Metadaten wie Homepage und der URL für den Quellcode. Die Version wird in Spack als Tupel bestehend aus einem Versions-String und einem Hashwert zur eindeutigen Zuordnung dargestellt.

Abhängigkeiten Abhängigkeiten werden als Relationen der Form `depends_on(<dependency>)` bzw. `depends_on(<dependency>, <config>)` dargestellt, wobei letztere die oben genannten Suffixe zur Spezifizierung von Paketkonfigurationen als zweiten Parameter akzeptiert.
Von diesen Relationen können je nach Anzahl der Abhängigkeiten beliebig viele oder auch gar keine vorhanden sein.

Installation Der für die Installation entscheide Teil, gewissermaßen das İnstallationsskript", ist die Funktion `install(self, spec, prefix)`, welche den Ablauf der Installation steuert. Standardmäßig ruft diese Funktion lediglich ein Makefile auf; einmal ohne angegebenes Ziel und einmal mit dem Ziel `install`, um die Software nach dem

Bauen zu installieren. Dies ist gängige Praxis in vielen Softwareprojekten. Es ist ebenfalls möglich, an dieser Stelle beleiebig komplexere Skripte zu implementieren.

Literaturverzeichnis

[Pro18] Exascale Computing Project. Spack: The deployment tool for ecp's software stack, 2018.

[TG19] Et al. Todd Gamblin. Spack documentation, 2019.