

Spackを使って複雑なHPCソフトウェア環境を管理しよう

PEARC21 終日チュートリアル


2021年7月19日

本スライドの最新バージョンは以下のリンクを参照のこと
<https://spack-tutorial.readthedocs.io>

※翻訳
2021年8月25日
株式会社工クサ 堀 扶

チュートリアル資料

- このスライドや関連スクリプト
 - spack-tutorial.readthedocs.io
- Spackチャットルーム (Slack)
 - slack.spack.io
 - [tutorialチャンネルへ参加のこと](#)
- Slackに一度参加するとハンズオンエクササイズのためのロビー認証が与えられる



Spack

latest

LINKS

Main Spack Documentation

TUTORIAL

Basic Installation Tutorial
Environments Tutorial
Configuration Tutorial
Package Creation Tutorial
Developer Workflows Tutorial
Mirror Tutorial
Stacks Tutorial

[Read the Docs](#) v: latest ▼

[Docs](#) » [Tutorial: Spack](#)

Tutorial: Spack

This is a full-day introductory virtual event at the 2019 Linux conference.

You can use these materials and read the live demo.

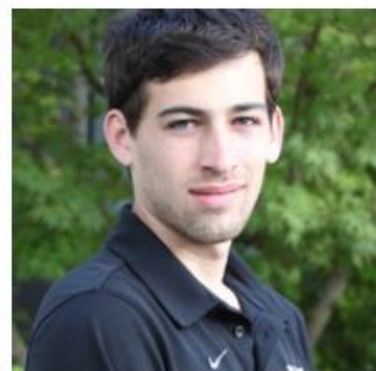
Slides



Complexity with Spack
Virtual event. July 19

Live Demos

チュートリアル提供者



Greg Becker
LLNL



Robert Blake
LLNL



Massimiliano
Culpo
np-complete, S.r.l.



Tamara
Dahlgren
LLNL



Adam Stewart
UIUC



Todd Gamblin
LLNL

アジェンダ

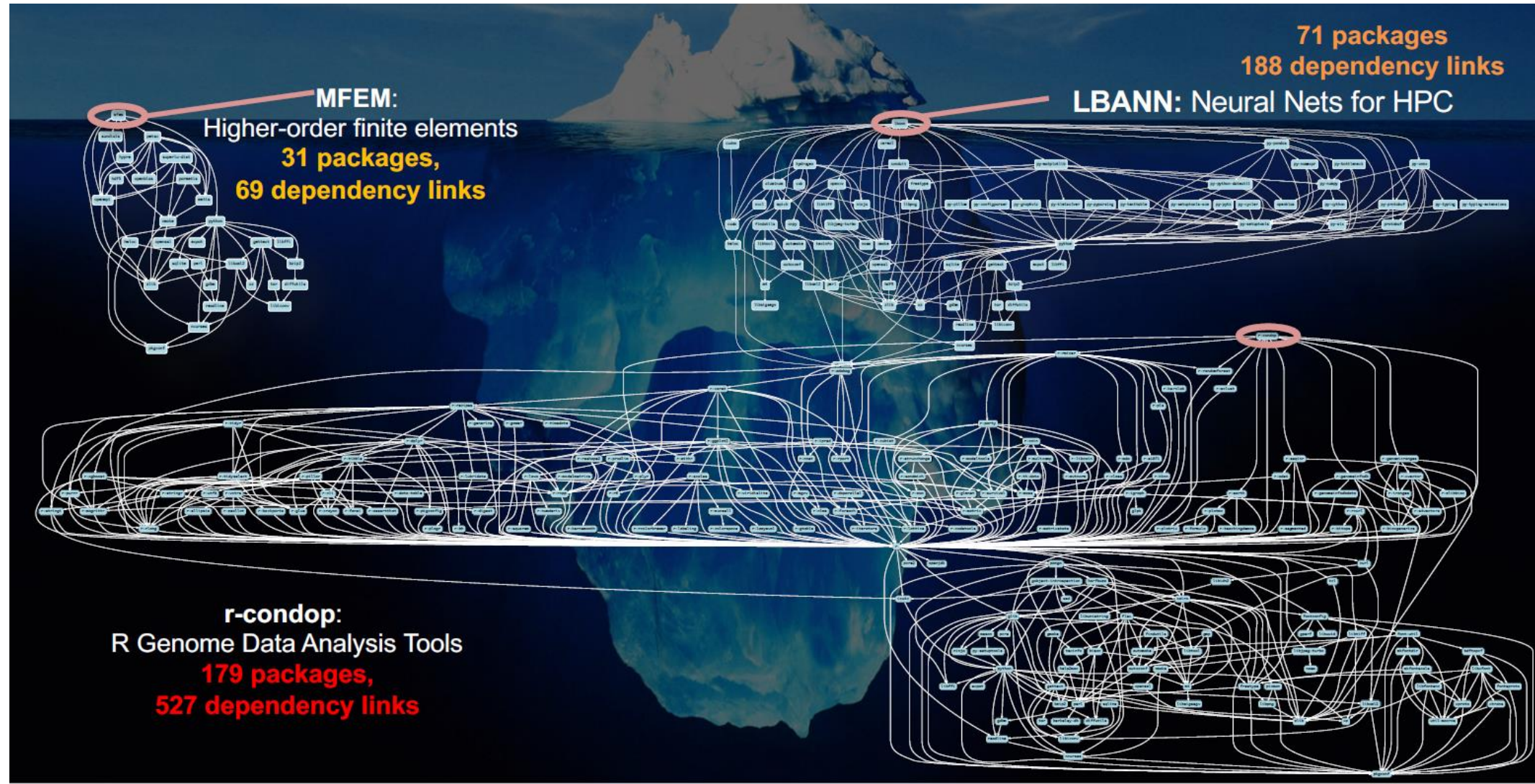
午前

・ イントロ	10分
・ 基礎	45分
・ コンセプト	20分
休憩	30分
・ 環境	45分
・ 設定	45分
休憩	30分

午後

・ パッケージング	60分
・ 開発ワークフロー	30分
休憩	30分
・ ミラー	20分
・ スタック	20分
・ スクリプト	15分
・ ロードマップ	10分

現代の科学技術計算コードは氷山のようなライブラリの依存関係に



パッケージ管理ツール(conda/pip/apt他) による一般的(懐疑的)な仮説

- ソースコードとバイナリは(プラットフォームごとに)1:1の関係
 - 再現性に優れている(例: Debian)
 - パフォーマンス最適化が難しい
- バイナリは可能な限り可搬性をもつべき
 - たいていのディストリビューションで動作
 - 再現可能だがパフォーマンスは悪い
- エコシステム間で同一の[ツールチェーン](#)
 - 単一のコンパイラ、単一のランタイムライブラリセット
 - もしくはコンパイラなし(インタプリタ言語)

HPCは多くの仮説に違反

- 一般的にコードはソースとして分離
 - ベンダライブラリ、コンパイラを除く
- 同一ライブラリなのに多くのバリエーションビルド
 - 開発者ごとに異なるビルド
 - 新規マシンの場合多くが初回ビルド
- コードはプロセッサやGPUごとに最適化
 - メイクがハードウェアごとに影響
 - 10~100倍パフォーマンスが変わる
- システムパッケージに深く関連
 - マシンごとに最適化されたライブラリ使用が必要
 - ホストGPUライブラリおよびネットワーク使用が必要
- 複数の言語
 - C、C++、Fortran、Pythonほか
 - すべて同一のエコシステムに

現在



Oak Ridge National Lab
Power9 / NVIDIA



RIKEN
Fujitsu/ARM a64fx

今後



Lawrence Berkeley
National Lab
AMD Zen / NVIDIA



Argonne National Lab
Intel Xeon / Xe



Oak Ridge National Lab
AMD Zen / Radeon



Lawrence Livermore
National Lab
AMD Zen / Radeon

コンテナ

- コンテナは、構築済みソフトウェアスタックの複製・配布に優れている
- 誰かがコンテナをビルドする必要がある！
- 些細なことではない
- コンテナ化されたアプリは数百の関連を保持
- コンテナ内でのOSパッケージマネージャ使用は不十分
- たいていのバイナリビルドは最適化されていない
- 一般的なライブラリはアーキテクチャ固有の最適化がおこなわれていない
- HPCコンテナは多くの異なるホストでとりあえずリビルドが必要
- すべてのファシリティに一つのコンテナが構築できるか明確になっていない
- コンテナはHPCにおけるNプラットフォーム問題を解決できない可能性がある

コンテナをビルドするためのより柔軟な手段が必要



SpackがHPCにおけるソフトウェア配布を可能に

- Spackは科学技術計算向けソフトウェアのビルド・インストールを自動化
- パッケージはパラメータ化されているため、ユーザはかんたんに構成を調整可能

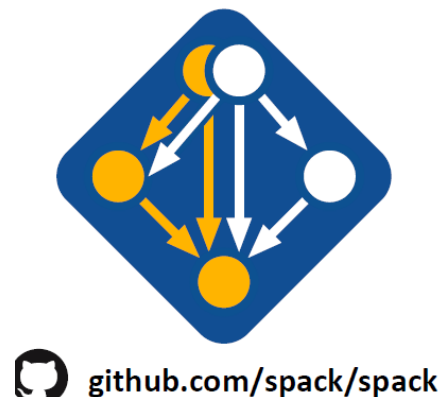
インストール不要: cloneして実行するだけ

```
$ git clone https://github.com/spack/spack  
$ spack install hdf5
```

単純な文法で複雑なインストールを可能に

```
$ spack install hdf5@1.10.5  
$ spack install hdf5@1.10.5 %clang@6.0  
$ spack install hdf5@1.10.5 +threadssafe  
$ spack install hdf5@1.10.5 cppflags="-O3 -g3"  
$ spack install hdf5@1.10.5 target=haswell  
$ spack install hdf5@1.10.5 +mpi ^mpich@3.2
```

- 利用が簡単なメインストリームツール、HPC環境で柔軟性も維持
- それに加えCLIで
- モジュールの生成(必須ではない)
- conda/virtualenvライクな環境
- 多くのDevOps機能を提供(CI、コンテナ生成ほか)



誰がSpackを使うの？

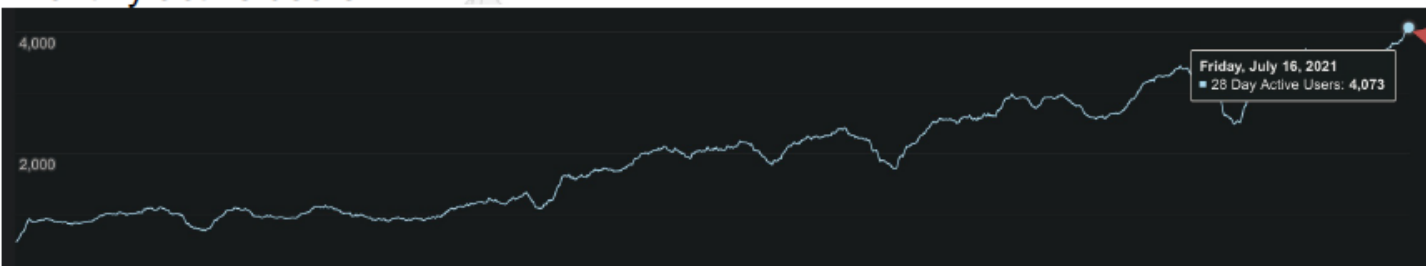
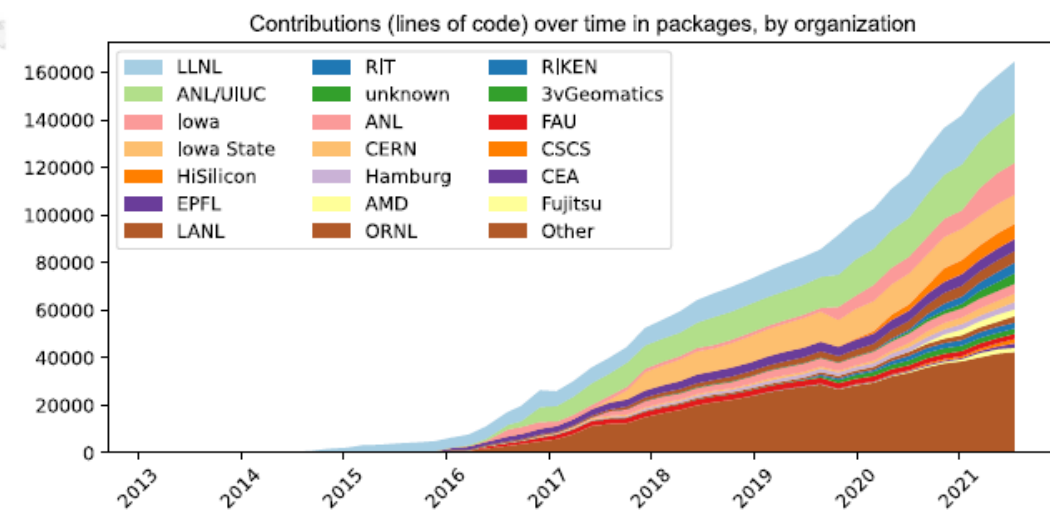
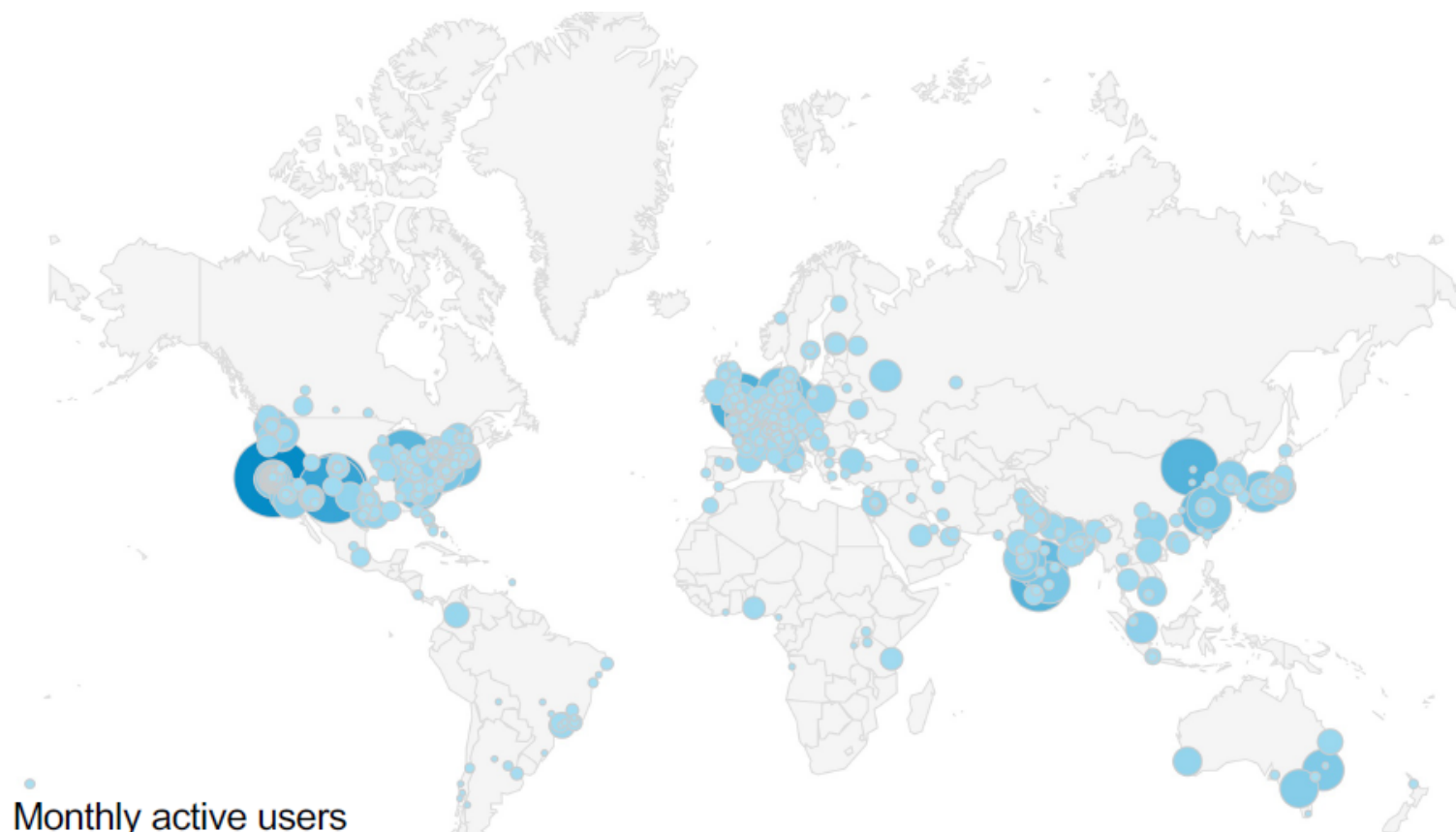
HPC用のソフトウェアを使いたい人！

- HPCソフトウェアのエンドユーザ
 - HPCアプリやツールをインストールし実行する
- HPCアプリケーションチーム
 - サードパーティ関連ライブラリの管理
- パッケージ開発者
 - 各自のソフトウェア配布に必要なパッケージが欲しい人
- HPCコンテナユーザサポートチーム
 - 大規模HPCサイトでのユーザのためのソフトウェア配置

成長中のSpackコミュニティ

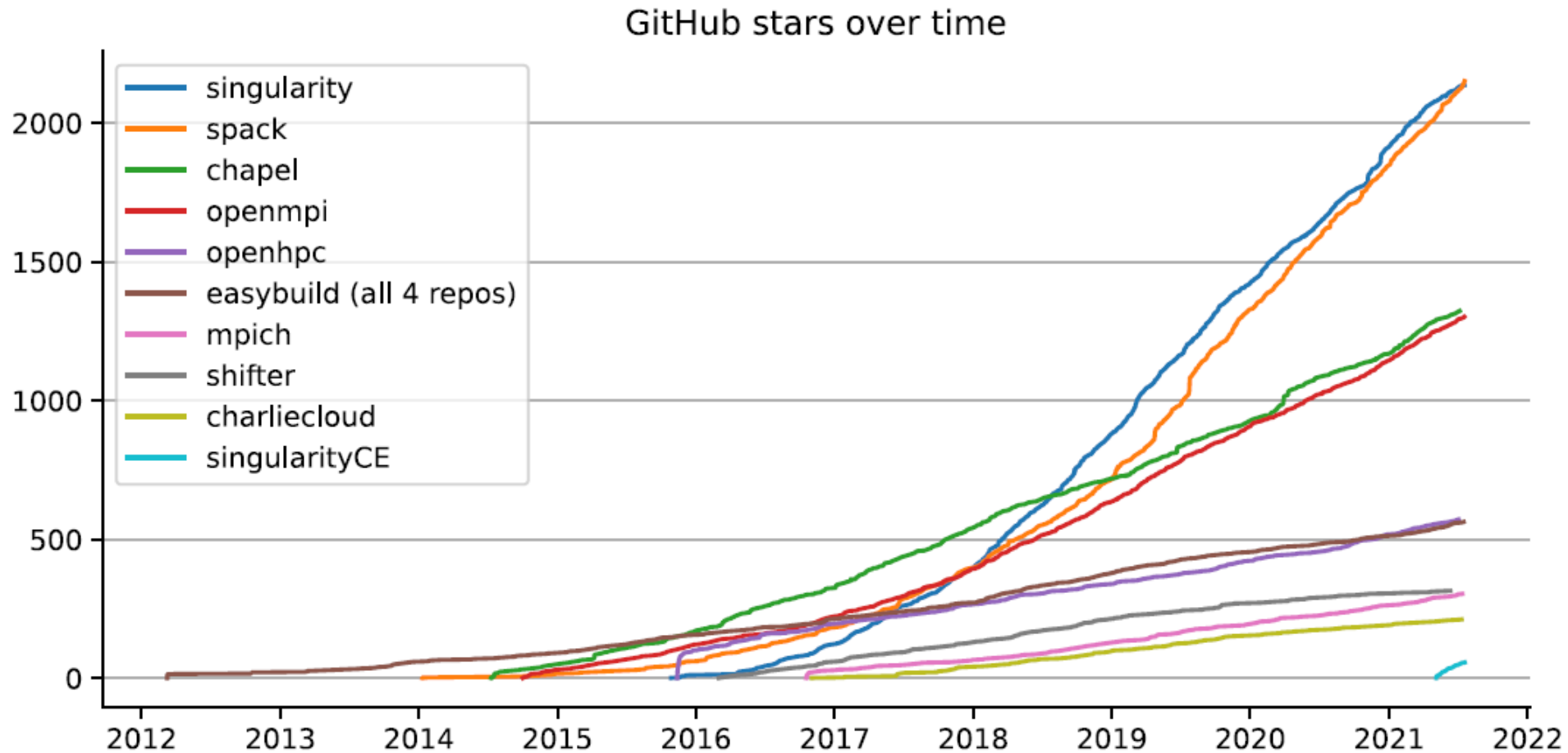
5700を超えるソフトウェアパッケージ
840の貢献者

570パッケージ提供率
2020年も増加



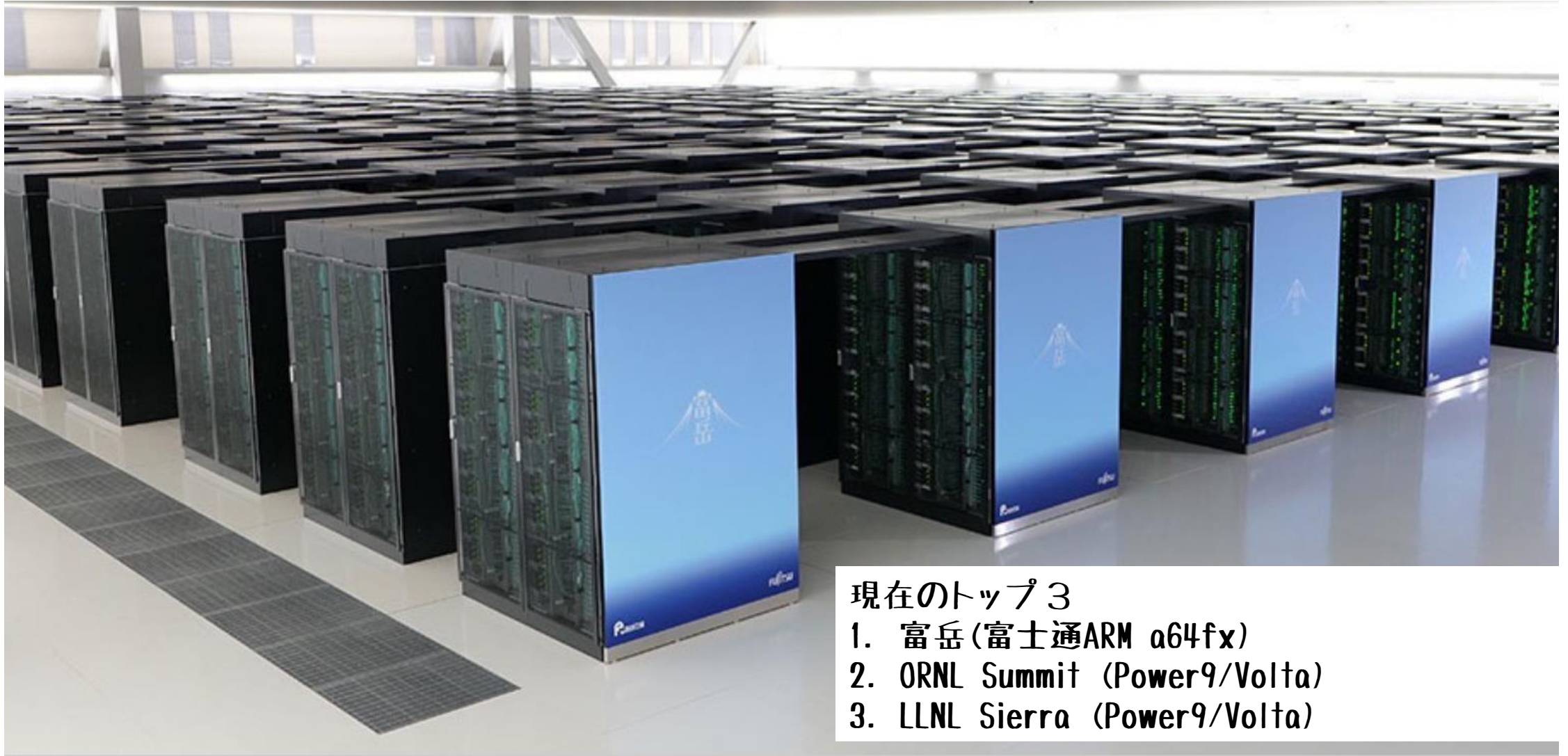
先週の月間アクティブユーザ数は
史上最高の4,073人

Spackは急速に採用されている(★を指標とした場合)



もしこのチュートリアルで気に入ったらgithub.com/spack/spack に★をつけてね

Spackは世界一のスパコンに採用されている



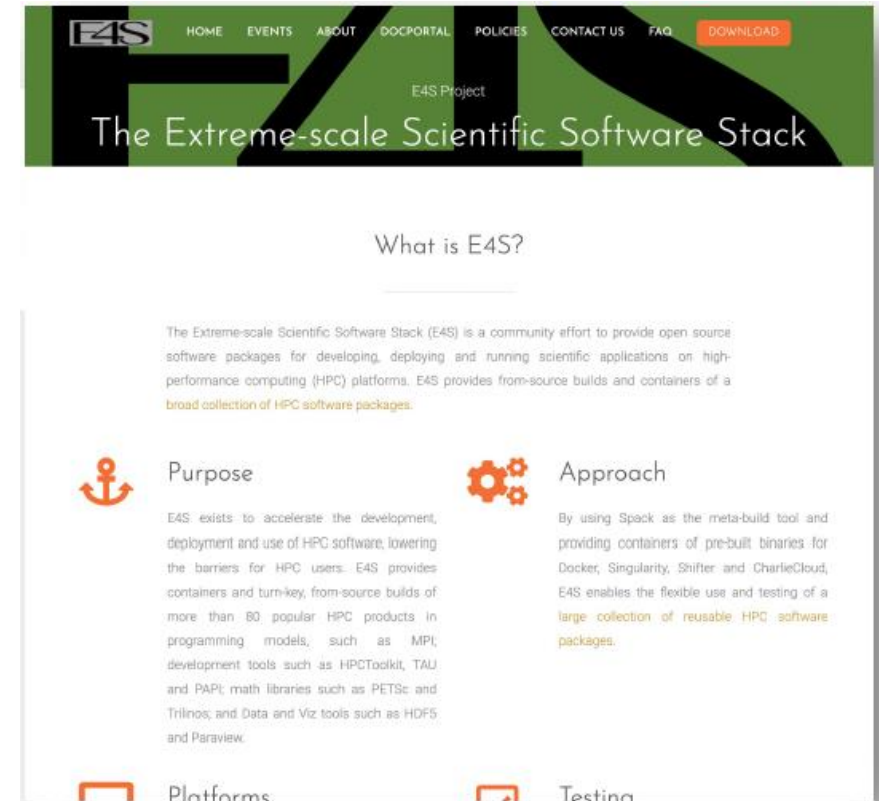
現在のトップ3

1. 富岳(富士通ARM a64fx)
2. ORNL Summit (Power9/Volta)
3. LLNL Sierra (Power9/Volta)

Spackは米エクサスケールコンピューティングプロジェクトの配置ツール



- Spackは今後3つの米エクサスケールシステム用ソフトウェア構築に使用されます
- ECPLはSpackを使ってExtreme Scale Scientific Software Stack (E4S)をビルドしました（詳細は<https://e4s.io>を参照）
- 我々はECPLにおける堅牢かつ有能なエクサスケールソフトウェアエコシステム作成ミッションを支援しています

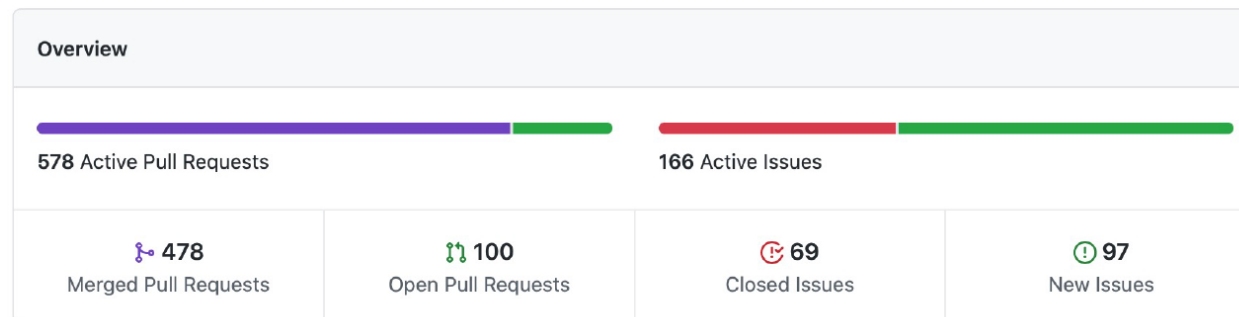


<https://e4s.io>

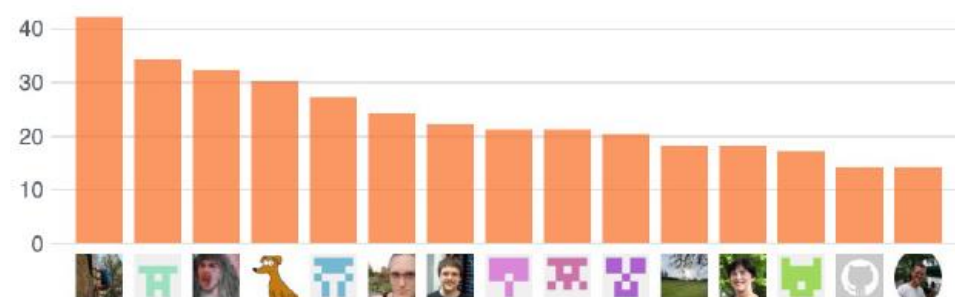
ここ1ヶ月Spack開発はとても忙しい！

April 20, 2021 – May 20, 2021

Period: 1 month ▾



- マージを除き、147人の作成者が開発のために467のコミットをプッシュし、566のコミットをすべてのブランチにプッシュ
- ある1つの開発では、596ファイルが変更、8,995ファイルが追加、3,311ファイルが削除された



Spackへの企業からの貢献も増加

- 富士通および理研は膨大な富岳ARM/a64fx用パッケージに貢献
- ARMはROCmパッケージやコンパイラサポートに貢献
 - ARMを主とした55以上のプルリクほか
 - ROCm、HIP、aoccパッケージはすべてSpackに取込
- Intelは我々のビルドファームのためにoneapiサポートやコンパイラライセンスに貢献
- NVIDIAはNVHPCコンパイラサポートやその他機能に貢献
- ARMおよびLinaroメンバはARMサポートに貢献
 - 400以上のARMサポートのための様々なコンパイラのプルリクエスト
- AWSは我々のビルドファームと協業し、ParallelClusterのための最適化バイナリビルド
 - 7月のAWSジョイントSpackチュートリアルに125名以上が参加

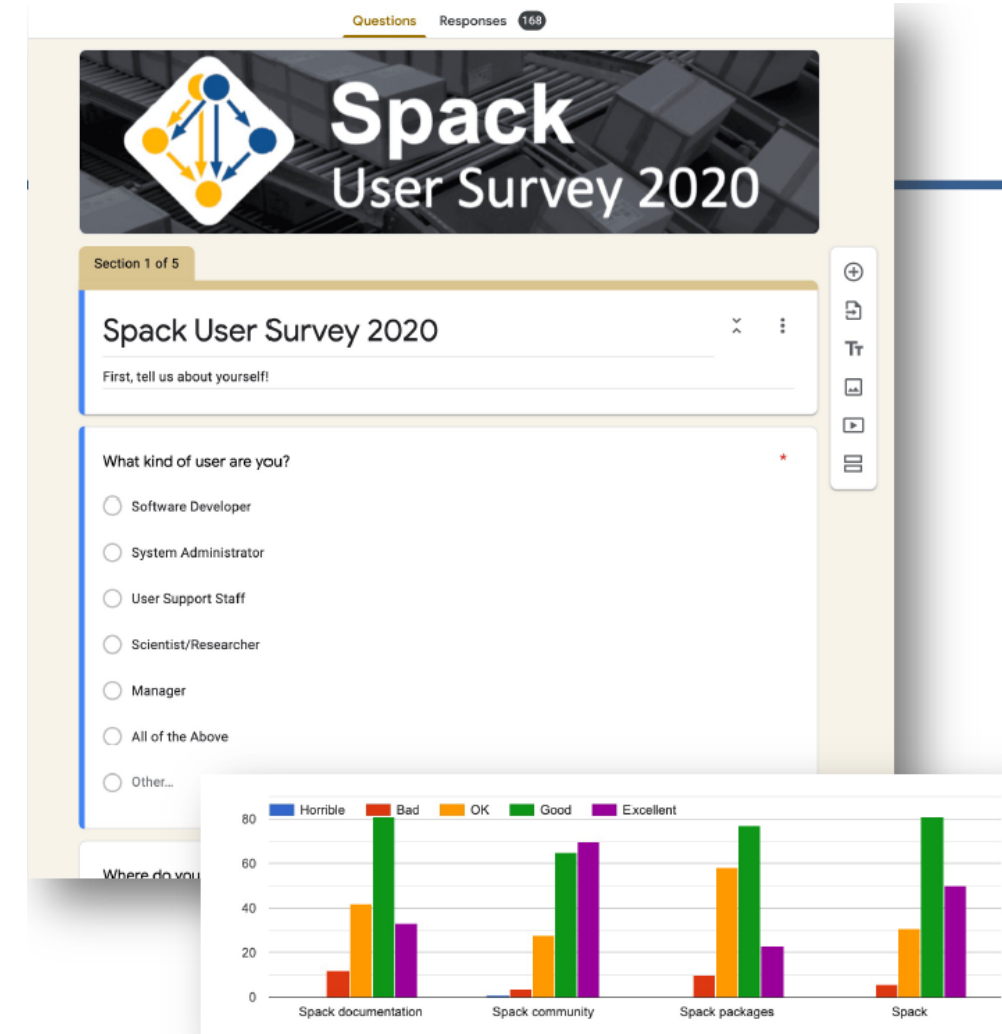


Spackユーザ調査2020

- Spackとしては初めての大規模調査
- Slackの全チャンネルへ送信(900ユーザ以上)
- すべてのSpackメーリングリスト、ECPメーリングリスト
- 169件の返信、以下持ち帰り
 - Spackとそのコミュニティが好き
 - ドキュメントやパッケージの安定にむけてもっと多くの作業が必要
 - コンクリート化機能と開発機能は最も求められている改善点

結果詳細：

<https://spack.io/spack-user-survey-2020>



Spackは唯一のビルド自動化ツールではない

- 「機能的な」パッケージマネージャ
 - Nix
 - GNU Guix
- ソースからビルドするパッケージマネージャ
 - Homebrew, LinuxBrew
 - MacPorts
 - Gentoo

HPC領域でのその他ツール

- Easybuild
 - HPC用インストールツール
 - HPCシステム管理にフォーカスーSpackとは異なるパッケージモデル
 - 固定されたソフトウェアスタックに依存ー実験用レシピ調整が難しい
- Conda
 - データサイエンス用バイナリパッケージマネージャとしてはとても有名
 - HPCだけがターゲットではない；一般的に最適化されていないバイナリ

<https://nixos.org>

<https://www.gnu.org/s/guix/>

<https://brew.sh>

<https://www.macports.org>

<https://gentoo.org>

<http://hpcugent.github.com.io/easybuild/>

<https://conda.io>



The Conda logo, featuring a green circular icon with a white 'C' inside, followed by the word "CONDA" in a bold green sans-serif font.

ハンズオン : Spack基礎

フォロースクリプトは `spack-tutorial.readthedocs.io`

コア Spack の概念

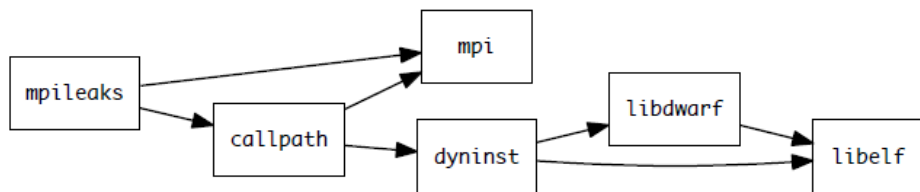
- まだslack上で我々に参加していない場合は、ここから招待を受け、チュートリアルチャンネルに参加し、VMログイン方法を効いてください
 - Slack slack.spack.io の #tutorial チャンネルに参加
- こちらのチュートリアルに従ってください
 - マテリアル spack-tutorial.readthedocs.io

ほとんどの既存ツールはバージョンの組み合わせをサポートしていない

- 典型的なバイナリパッケージマネージャ
 - RPM、yum、APT、yastなど
 - 単一スタック管理用に設計されている
 - 単一のプレフィックス(/usr)に単一のバージョンのパッケージをインストール
 - 十分にテストされた安定したスタックのシームレスな更新
- ポートシステム
 - BSD Ports、portage、Macports、Homebrew、Gentooなど
 - コンパイラや関連バージョンによるパラメータ化されたビルドを最小限サポート
- 仮想マシンおよびLinuxコンテナ(Docker)
 - コンテナはユーザに異なるアプリケーションのビルドを許可
 - ビルド問題を解決しない(誰かがイメージをビルドしなくてはならない)
 - パフォーマンス、セキュリティ、アップグレードなどの問題によりHPCの広範な配置を妨げる

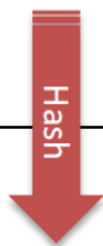
Spackは複雑なソフトウェア組み合わせを処理

Dependency DAG



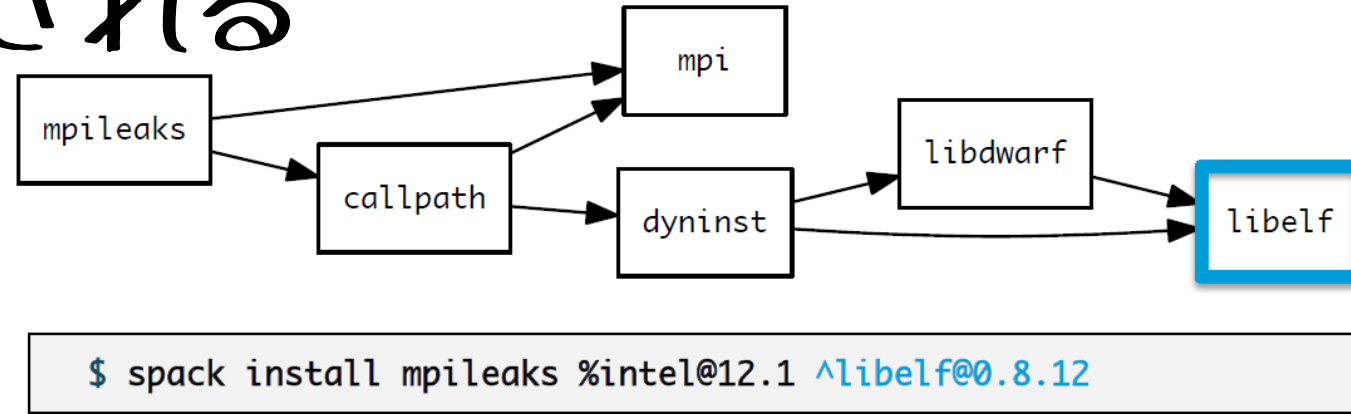
Installation Layout

opt
└─ spack
 ├─ darwin-mojave-skylake
 │ ├─ clang-10.0.0-apple
 │ │ ├─ bzip2-1.0.8-hc4sm4vuzpm4znmvrfzri4ow2mkphe2e
 │ │ ├─ python-3.7.6-daqqpssxb6qbfrztsezkmhus3xoflbsy
 │ │ ├─ sqlite-3.30.1-u64v26igxvyn23hysmklfums6tgjv5r
 │ │ ├─ xz-5.2.4-u5eawkvaoc7vonabe6nndkcfwuv233cj
 │ │ └─ zlib-1.2.11-x46q4wm46ay4pltrijbgizxjrhbaka6
 ├─ darwin-mojave-x86_64
 │ ├─ clang-10.0.0-apple
 │ └─ coreutils-8.29-pl2kcytej qcys5dzecfrtjqx fdssvnob



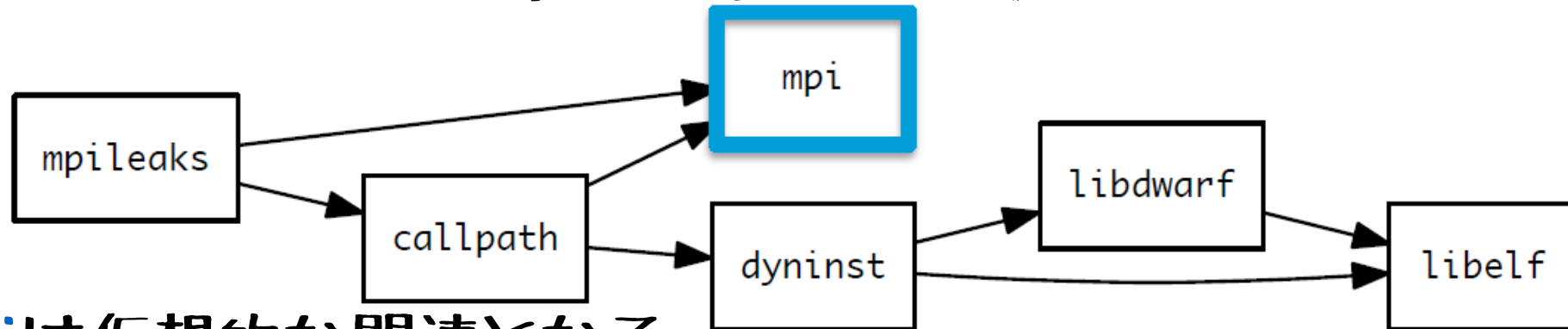
- 各ユニーク関連グラフが一意的な構成となる
- 一意的なディレクトリにそれぞれの構成
- 同一パッケージの複数の構成が共存可能
- 完全有向非巡回グラフ (DAG) のハッシュが各プレフィックスに追加
- インストール済みパッケージは自動的に関連を検索
 - Spackはバイナリ内にRPATHを埋め込む
 - モジュールの使用や環境変数LD_LIBRARY_PATHの設定は不要
 - 過去にあなたがビルドした方法で機能する

Spackの仕様により依存関係のバージョンが制約される



- SpackはDAGごとに各ライブラリに対する1つの構成を保証
 - [ABI](#)の一貫性を保証
 - ユーザはDAGの構造を知る必要がない；関連名のみでよい
- Spackは同一のコンパイラを使ったビルドを保証し、組み合わせることも可能
 - コンパイラが混在している場合のABI互換性を保証して動作

SpackはMPIなどABI非互換のバージョン管理されたインターフェイスを処理



- mpiは仮想的な関連となる
- 2つの異なるMPI実装を使ってビルドされた同じパッケージをインストール：

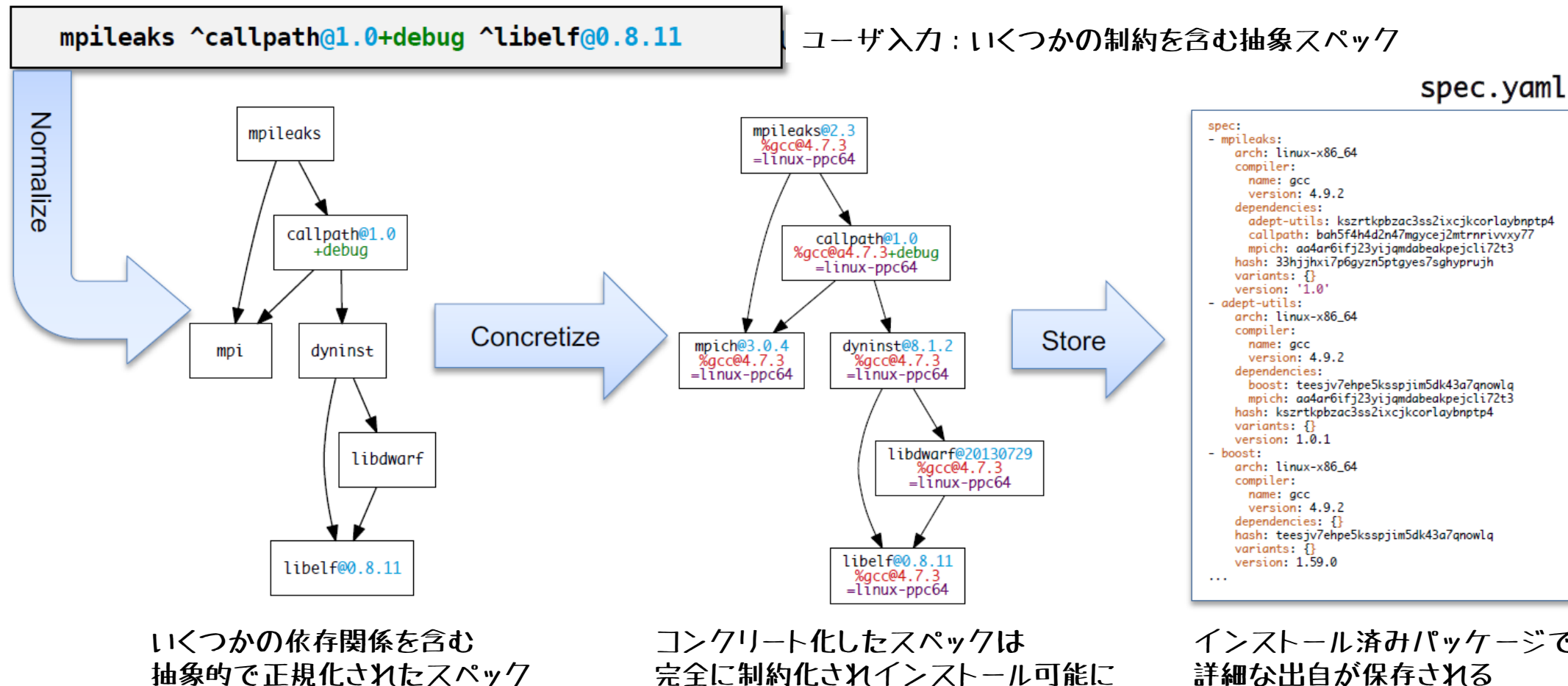
```
$ spack install mpileaks ^mvapich@1.9
```

```
$ spack install mpileaks ^openmpi@1.4:
```

- Spack は、MPI2インターフェイスを提供するMPI実装を選択

```
$ spack install mpileaks ^mpi@2
```


ユーザが明示的ではない場合、コンクリート化により不足した構成詳細が埋められる

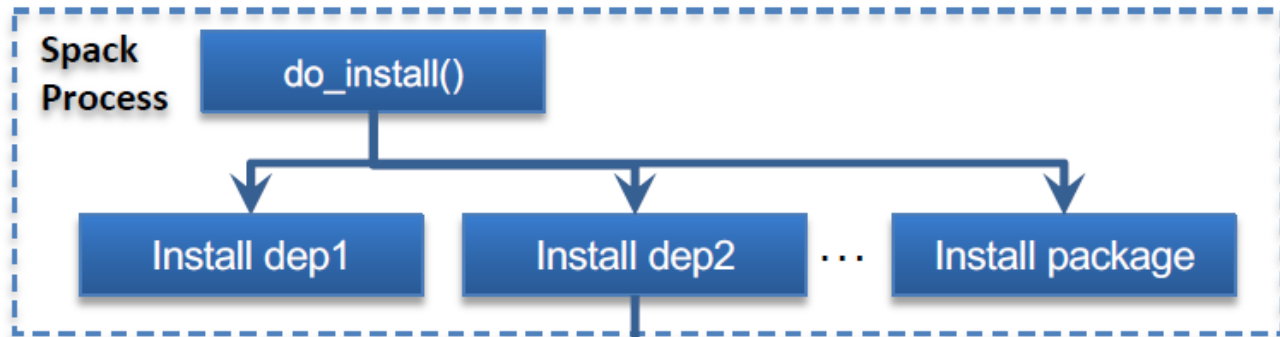


spack specコマンドによりコンクリート化された結果を参照

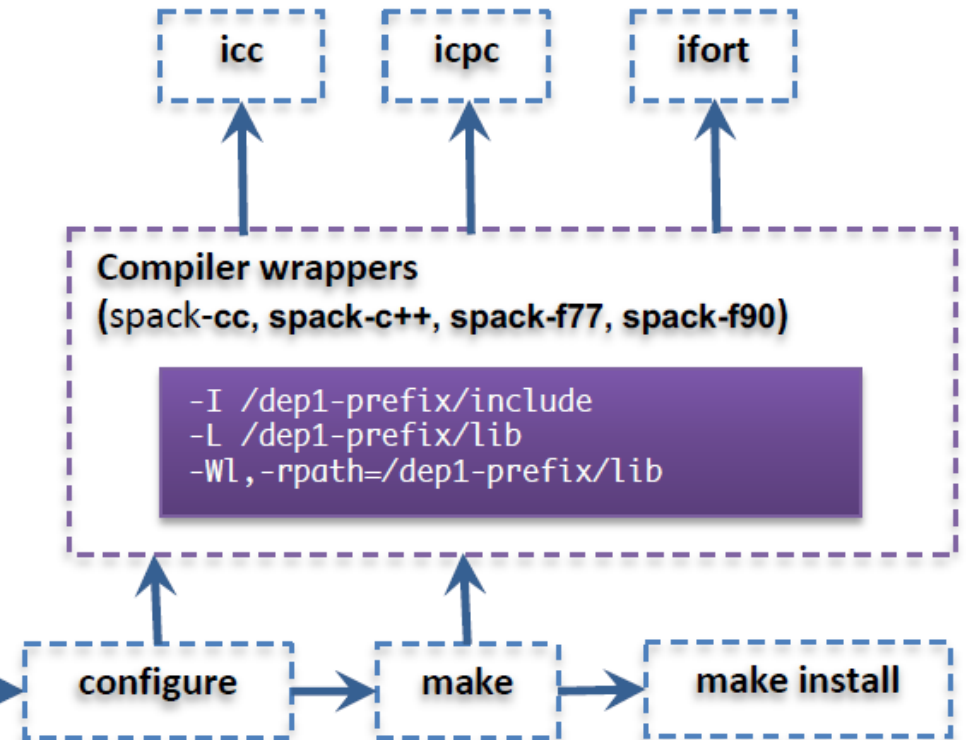
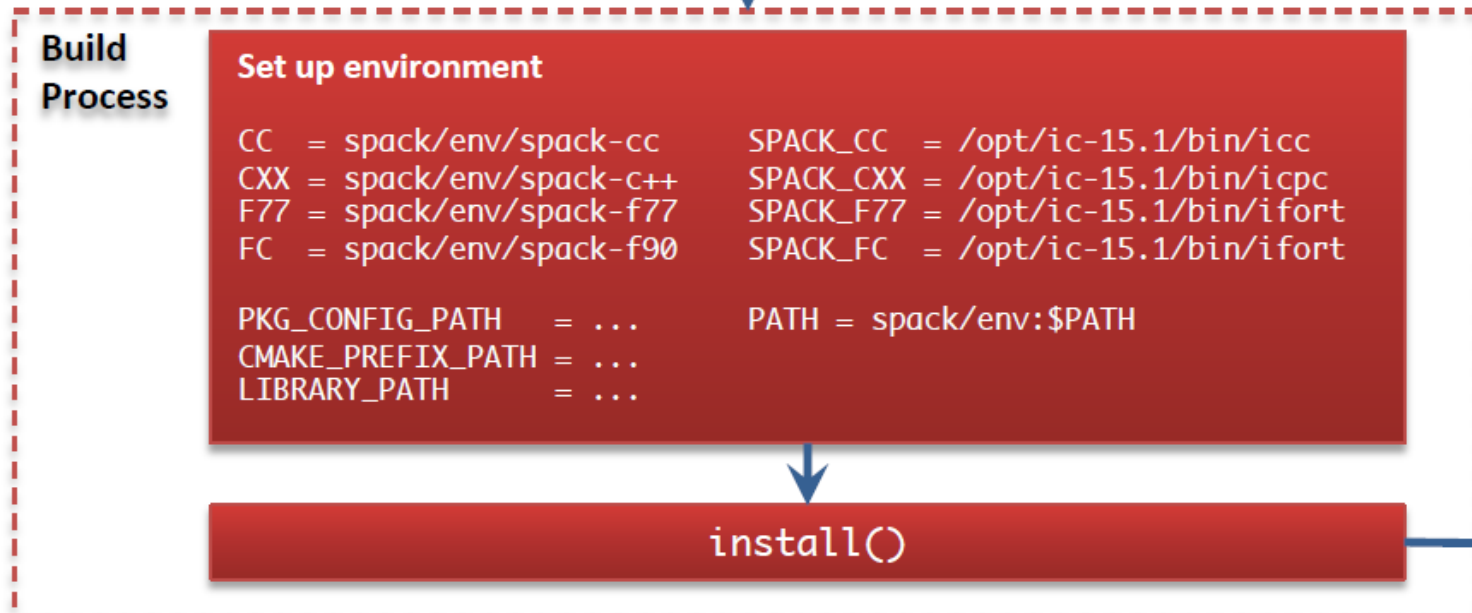
```
$ spack spec mpileaks
Input spec
-----
mpileaks

Concretized
-----
mpileaks@1.0%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^adept-utils@1.0.1%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^boost@1.61.0%gcc@5.3.0+atomic+chrono+date_time~debug+filesystem~graph
~icu_support+iostreams+locale+log+math~mpi+multithreaded+program_options
~python+random +regex+serialization+shared+signals+singlethreaded+system
+test+thread+timer+wave arch=darwin-elcapitan-x86_64
^bzip2@1.0.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^zlib@1.2.8%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^openmpi@2.0.0%gcc@5.3.0~mxm~pmi~psm~psm2~slurm~sqlite3~thread_multiple~tm~verbs+vt arch=darwin-elcapitan-x86_64
^hwloc@1.11.3%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^libpciaccess@0.13.4%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^libtool@2.4.6%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^m4@1.4.17%gcc@5.3.0+sigsegv arch=darwin-elcapitan-x86_64
^libsigsegv@2.10%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^callpath@1.0.2%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^dyninst@9.2.0%gcc@5.3.0~stat_dysect arch=darwin-elcapitan-x86_64
^libdwarf@20160507%gcc@5.3.0 arch=darwin-elcapitan-x86_64
^libelf@0.8.13%gcc@5.3.0 arch=darwin-elcapitan-x86_64
```

Spackは独自のコンパイル環境で各パッケージをビルド



- フォークされたビルド処理は、環境を分離
コンパイララップを使用する：
 - include、lib、RPATHフラグを追加
 - 依存関係は自動的に検索
 - Crayモジュールのロード(正しいコンパイラ/システム依存関係)



ExtentionおよびPythonサポート

- Spackはパッケージを各々のプレフィックス内にインストール
- いくつかのパッケージは他のパッケージのディレクトリ構造内にインストールされる必要がでてくる
 - 例えば、\$prefix/lib/python-<バージョン>/site-packages にインストールされたPythonモジュール
 - Spackはextensionを経由してサポート

```
class PyNumpy(Package):  
    """NumPy is the fundamental package for scientific computing with Python."""  
  
    homepage = "https://numpy.org"  
    url      = "https://pypi.python.org/packages/source/n/numpy/numpy-1.9.1.tar.gz"  
    version('1.9.1', '78842b73560ec378142665e712ae4ad9')  
  
    extends('python')  
  
    def install(self, spec, prefix):  
        setup_py("install", "--prefix={0}".format(prefix))
```

Spack extensions


- いくつかのパッケージは他のパッケージのディレクトリ構造内にインストールされる必要がある
- 拡張パッケージの例：
 - python ライブラリが良い例
 - R、Lua、perl
 - バージョンの組み合わせも維持する必要あり

```
$ spack activate py-numpy @1.10.4
```

- Spackインストール場所へシンボリックリンク
- これは古い機能ですー今後はspack environments を代わりに使用することになります
 - 詳細は後述！

```
spack/opt/  
  linux-rhel6-x86_64/  
    gcc-4.7.2/  
      python-2.7.12-6y6vvaw/  
        lib/python2.7/site-packages/  
          ..  
            py-numpy-1.10.4-oaix36/  
              lib/python2.7/site-packages/  
                numpy/  
          ...
```

```
spack/opt/  
  linux-rhel6-x86_64/  
    gcc-4.7.2/  
      python-2.7.12-6y6vvaw/  
        lib/python2.7/site-packages/  
          numpy@  
            py-numpy-1.10.4-oaix36/  
              lib/python2.7/site-packages/  
                numpy/  
          ...
```



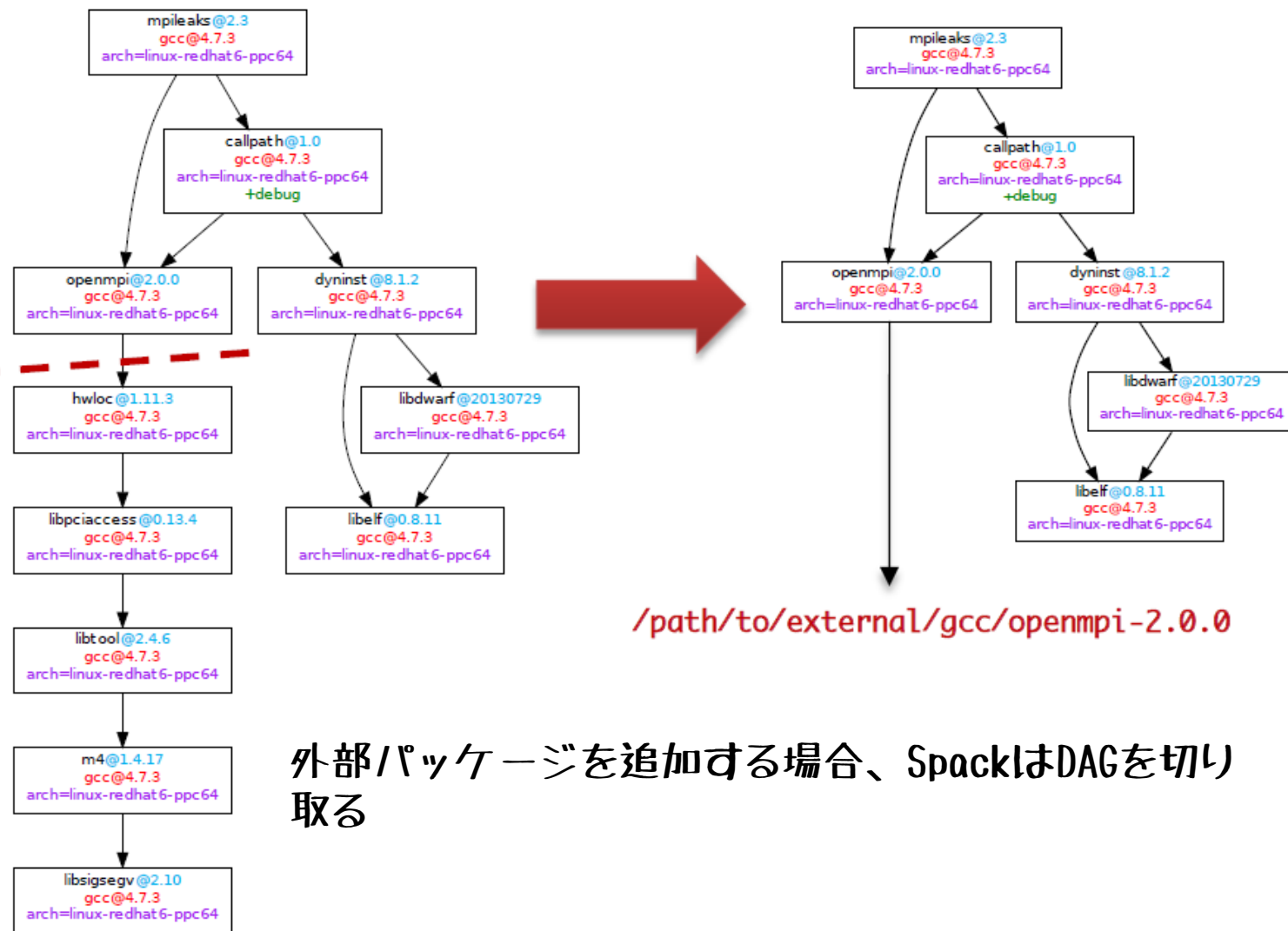
外部インストールされたソフトウェアに対するビルド

```
mpileaks ^callpath@1.0+debug  
^openmpi ^libelf@0.8.11
```

packages.yaml

```
packages:  
  mpi:  
    buildable: False  
    paths:  
      openmpi@2.0.0 %gcc@4.7.3 arch=linux-rhel6-ppc64:  
        /path/to/external/gcc/openmpi-2.0.0  
      openmpi@1.10.3 %gcc@4.7.3 arch=linux-rhel6-ppc64:  
        /path/to/external/gcc/openmpi-1.10.3  
      ...
```

ユーザは設定ファイル内に外部パッケージを登録する（詳細は後述）



Spackパッケージリポジトリ

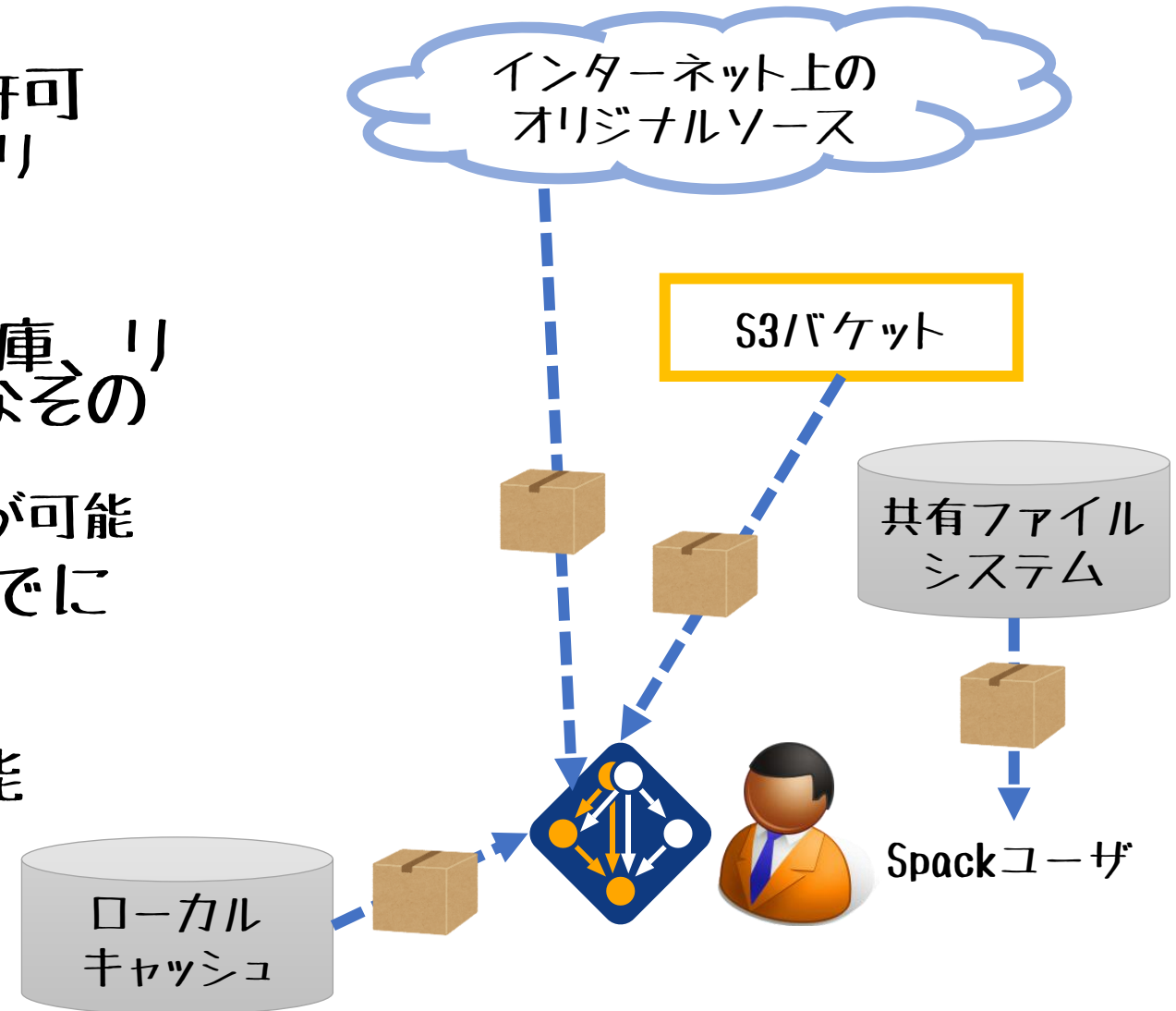
- Spackは外部パッケージリポジトリをサポートしている
- パッケージレシピのディレクトリを分割する
- 使用する理由：
 - 一部のパッケージを公開したくない
 - いくつかのサイトが奇抜なカスタムビルドを必要とする
 - サイト固有のバージョンでデフォルトパッケージをオーバーライドしたい
- パッケージは構成可能：
 - 外部リポジトリはビルトインパッケージのトップに階層化可能
 - カスタムパッケージはビルトインパッケージ(もしくは他のリポジトリ内のパッケージ)に依存可能

```
$ spack repo create /path/to/my_repo
$ spack repo add my_repo
$ spack repo list
==> 2 package repositories.
my_repo      /path/to/my_repo
builtin      spack/var/spack/repos/builtin
```



Spack ミラー

- Spackはユーザにミラー定義を許可
 - ファイルシステム内のディレクトリ
 - Webサーバ上
 - S3バケット内
- ミラーは、フェッチされたtar書庫、リポジトリ、およびビルドに必要なその他のリソースのアーカイブ
 - バイナリパッケージも含めることが可能
- デフォルトでは Spackはこれまでにフェッチした全てのミラーを `var/spack/cache` に保持
- サイト内部にミラーをホスト可能
 - 詳細はドキュメントを参照のこと

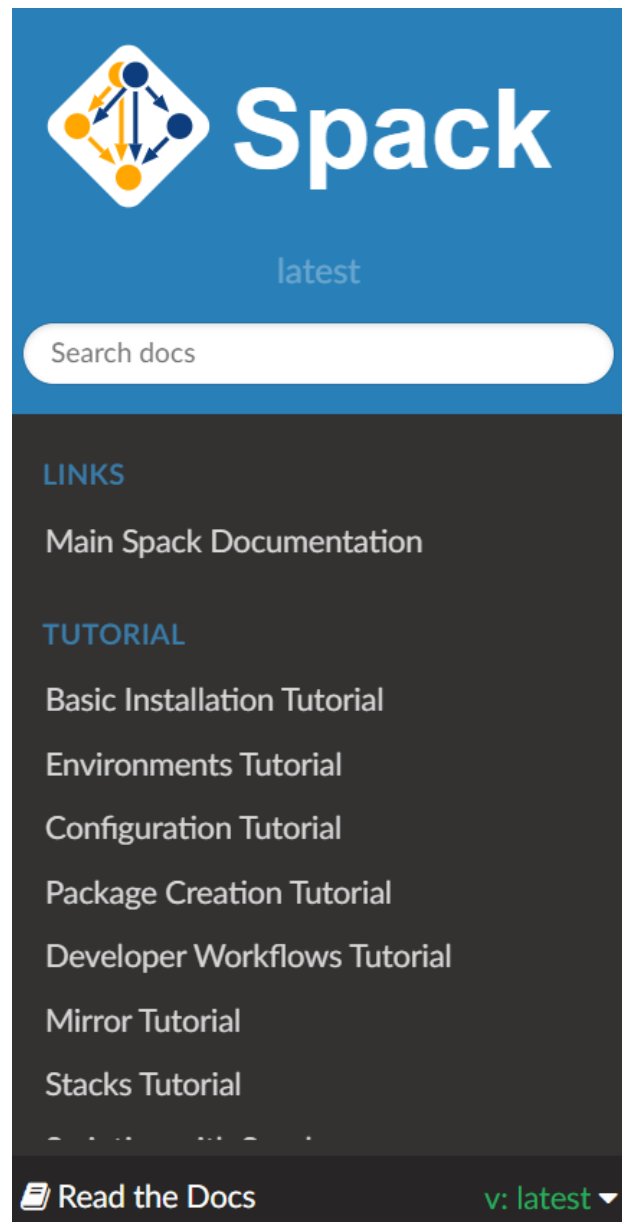


環境、`spack.yaml`、および`spack.lock`

フォロースクリプトは `spack-tutorial.readthedocs.io`

中央欧標準時午後4:20(太平洋夏時間午前7:20)に再開

- このスライドや関連スクリプト
 - spack-tutorial.readthedocs.io
- Spackチャットルーム (Slack)
 - slack.spack.io
 - [tutorial](#)チャンネルへ参加のこと
- Slackに一度参加するとハンズオンエクササイズのためのログイン認証が与えられる



Spack
latest

Search docs

LINKS

- Main Spack Documentation

TUTORIAL

- Basic Installation Tutorial
- Environments Tutorial
- Configuration Tutorial
- Package Creation Tutorial
- Developer Workflows Tutorial
- Mirror Tutorial
- Stacks Tutorial

[Read the Docs](#) v: latest ▼

[Docs](#) » [Tutorial: Spack](#)

Tutorial: Spack

This is a full-day introductory virtual event at the 2019 conference.

You can use these materials and read the live demo.

Slides



Complexity with Spack
Virtual event. July 19


Live Demos

ハンズオン：構成

フォロースクリプトは spack-tutorial.readthedocs.io

チュートリアル資料

- このスライドや関連スクリプト
 - spack-tutorial.readthedocs.io
- Spackチャットルーム (Slack)
 - slack.spack.io
 - [tutorialチャンネルへ参加のこと](#)
- Slackに一度参加するとハンズオンエクササイズのためのロギン認証が与えられる



Spack

latest

LINKS

Main Spack Documentation

TUTORIAL

Basic Installation Tutorial
Environments Tutorial
Configuration Tutorial
Package Creation Tutorial
Developer Workflows Tutorial
Mirror Tutorial
Stacks Tutorial

[Read the Docs](#) v: latest ▼

[Docs](#) » [Tutorial: Spack](#)

Tutorial: Spack

This is a full-day introductory virtual event at the 2019 Linux conference.

You can use these materials and read the live demo.

Slides



Complexity with Spack
Virtual event. July 19

Live Demos

ハンズオン：パッケージ作成

フォロースクリプトは spack-tutorial.readthedocs.io

ハンズオン：開発者ワークフロー

フォロースクリプトは spack-tutorial.readthedocs.io

ハンズオン：バイナリキャッシュおよび ミラー

フォロースクリプトは spack-tutorial.readthedocs.io

ハンズオン：スタック

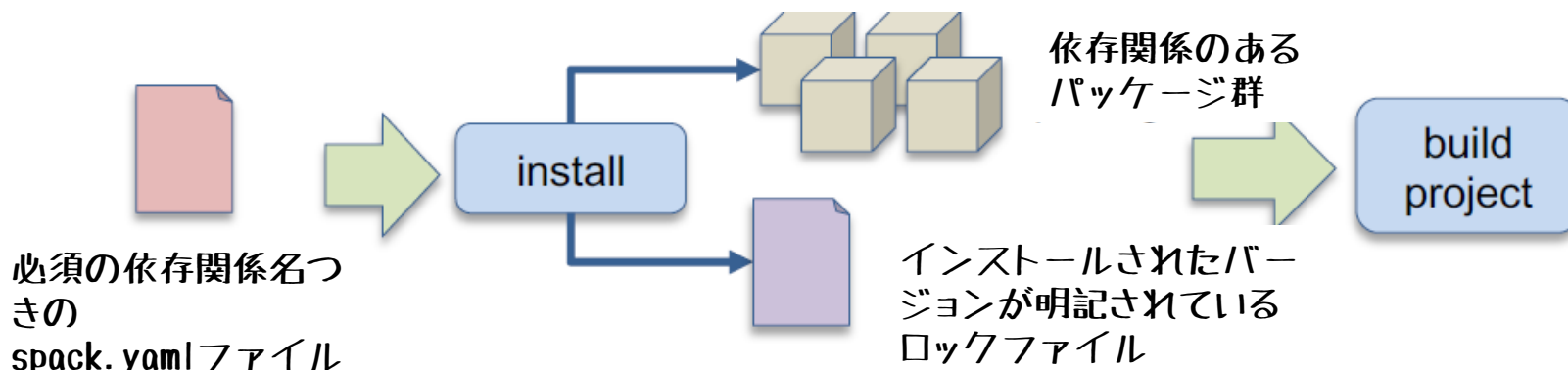
フォロースクリプトは spack-tutorial.readthedocs.io

ハンズオン：スクリプティング

フォロースクリプトは spack-tutorial.readthedocs.io

その他の機能と今後

Spack環境は複雑なワークフローの基礎



- 2つのファイル：
 - spack.yaml プロジェクト必要条件を記述
 - spack.lock インストール済みバージョン及び明確な構成を記録
 - 多くの構成の再現を可能に
- 以下のために環境を使用：
 - コンテナの構築 (spack containerize)
 - 継続的インテグレーションビルドの自動生成 (spack ci)
 - 配置 (matrix, spack stacks)
 - 開発者ワークフロー (新規機能)

```
spack:
  # include external configuration
  include:
    - ../special-config-directory/
    - ./config-file.yaml

  # add package specs to the `specs` list
  specs:
    - hdf5
    - libelf
    - openmpi
```

Concrete spack.lock file (generated)

```
{
  "concrete_specs": {
    "6s63so2kstp3zyvjzglmavy6l3nul": {
      "hdf5": {
        "version": "1.10.5",
        "arch": {
          "platform": "darwin",
          "platform_os": "mojave",
          "target": "x86_64"
        }
      },
      "compiler": {
        "name": "clang",
        "version": "10.0.0-apple"
      },
      "namespace": "builtin",
      "parameters": {
```



環境からコンテナイメージを生成(0.14)

```
spack:
  specs:
  - gromacs+mpi
  - mpich
```

```
container:
  # Select the format of the recipe
  # singularity or anything else
  format: docker
```

```
# Select from a valid list of images
base:
  image: "centos:7"
  spack: develop
```

```
# Whether or not to strip binaries
strip: true
```

```
# Additional system packages that
os_packages:
  - libgomp
```

```
# Extra instructions
extra_instructions:
  final: |
```

```
RUN echo "export PS1="\${(tput bold)}
```

```
# Labels for the image
labels:
  app: "gromacs"
  mpi: "mpich"
```

```
# Build stage with Spack pre-installed and ready to be used
FROM spack/centos7:latest as builder

# What we want to install and how we want to install it
# is specified in a manifest file (spack.yaml)
RUN mkdir /opt/spack-environment \
  && (echo "spack:" \
  && echo "  specs:" \
  && echo "    - gromacs+mpi" \
  && echo "    - mpich" \
  && echo "  concretization: together" \
  && echo "  config:" \
  && echo "    install_tree: /opt/software" \
  && echo "    view: /opt/view") > /opt/spack-environment/spack.yaml

# Install the software, remove unnecessary deps
RUN cd /opt/spack-environment && spack install && spack gc -y

# Strip all the binaries
RUN find -L /opt/view/* -type f -exec readlink -f '{}' \; | \
  xargs file -i | \
  grep 'charset=binary' | \
  grep 'x-executable|x-archive|x-sharedlib' | \
  awk -F: '{print $1}' | xargs strip -s

# Modifications to the environment that are necessary to run
RUN cd /opt/spack-environment && \
  spack env activate --sh -d . >> /etc/profile.d/z10_spack_environment.sh

# Bare OS image to run the installed executables
FROM centos:7

COPY --from=builder /opt/spack-environment /opt/spack-environment
COPY --from=builder /opt/software /opt/software
COPY --from=builder /opt/view /opt/view
COPY --from=builder /etc/profile.d/z10_spack_environment.sh /etc/profile.d/z10_spack_environment.sh

yum update -y && yum install -y epel-release && yum update -y
yum install -y libgomp \
  && rm -rf /var/cache/yum && yum clean all

RUN echo "export PS1="\${(tput bold)}\${(tput setaf 1)}\${(tput setaf 2)}\u\${(tput
```



spack containerize

- Spack環境はコンテナイメージにバンドル可能

- オプションのコンテナ章にて細かなカスタマイズを紹介

- 生成されたDockerfileは、マルチステージビルドを使って最終的なイメージサイズを最小化

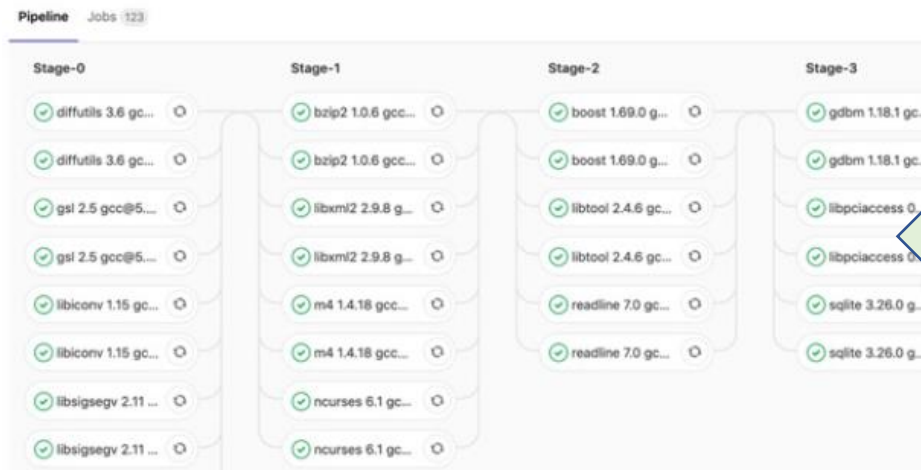
- バイナリの削除

- spack gc を使って不必要なビルド依存関係を除去

- Singularity レシピ生成も可能

Spackは環境からCIパイプラインを生成可能

- 環境に gitlab-ci セクションを追加
 - Spack はGitLabランナにビルドをマップ
 - spack ci コマンドで gitlab-ci.yaml を生成
- Kubeクラスター内やHPC環境のベアメタル上で実行可能
 - 進行状況を CDash webhook に送信



spack ci

```
spack:
  definitions:
    - pkgs:
      - readline@7.0
    - compilers:
      - '%gcc@5.5.0'
    - oses:
      - os=ubuntu18.04
      - os=centos7
  specs:
    - matrix:
      - [$pkgs]
      - [$compilers]
      - [$oses]
  mirrors:
    cloud_gitlab: https://mirror.spack.io
  gitlab-ci:
    mappings:
      - spack-cloud-ubuntu:
        match:
          - os=ubuntu18.04
        runner-attributes:
          tags:
            - spack-k8s
          image: spack/spack_builder_ubuntu_18.04
      - spack-cloud-centos:
        match:
          - os=centos7
        runner-attributes:
          tags:
            - spack-k8s
          image: spack/spack_builder_centos_7
  cdash:
    build-group: Release Testing
    url: https://cdash.spack.io
    project: Spack
    site: Spack AWS Gitlab Instance
```

spack external find

```
class Cmake(Package):
    executables = ['cmake']

    @classmethod
    def determine_spec_details(cls, prefix, exes_in_prefix):
        exe_to_path = dict(
            (os.path.basename(p), p) for p in exes_in_prefix
        )
        if 'cmake' not in exe_to_path:
            return None

        cmake = spack.util.executable.Executable(exe_to_path['cmake'])
        output = cmake('--version', output=str)
        if output:
            match = re.search(r'cmake.*version\s+(\S+)', output)
            if match:
                version_str = match.group(1)
                return Spec('cmake@{0}'.format(version_str))
```

package.py内に外部インストールコマンドを検出するためのロジックがある

```
packages:
  cmake:
    externals:
      - spec: cmake@3.15.1
        prefix: /usr/local
```

package.yaml 設定

- Spackはしばらくの間コンパイラ検出してきた
 - PATHからコンパイラを検出
 - 使用するために検出したコンパイラを登録
- 今では任意のパッケージの検出も可能
 - パッケージ定義:
 - 有効なコマンド名
 - コマンドの照会方法
 - Spackは既知コマンドを検出し設定に追加
- コミュニティでは簡単にセットアップするためのツールを提供

spack test: Spack/パッケージ内に直接テストを記述、ソフトウェア改善へ

```
class Libsigsegv(AutotoolsPackage, GNUMirrorPackage):
    """GNU libsigsegv is a library for handling page faults in user mode."""

    # ... spack package contents ...

    extra_install_tests = 'tests/.libs'

    def test(self):
        data_dir = self.test_suite.current_test_data_dir
        smoke_test_c = data_dir.join('smoke_test.c')

        self.run_test(
            'cc', [
                '-I%s' % self.prefix.include,
                '-L%s' % self.prefix.lib, '-lsigsegv',
                smoke_test_c,
                '-o', 'smoke_test'
            ],
            purpose='check linking')

        self.run_test(
            'smoke_test', [], data_dir.join('smoke_test.out'),
            purpose='run built smoke test')

        self.run_test('sigsegv1': ['Test passed'], purpose='check sigsegv1 output')
        self.run_test('sigsegv2': ['Test passed'], purpose='check sigsegv2 output')
```

テストは通常のSpackレシピクラスの一部になっている

パッケージから簡単にソースコードを保存

test() メソッドにテストを記述

保存済みソースを取得
実行形式へリンク

Spackはccを互換コンパイラと仮定

smoke test をビルドし妥当性を出力

パッケージでインストール済みプログラムを実行

spack developを使用することで開発者は一度で多くのパッケージで作業可能に

- これまで開発者は単一のパッケージにフォーカスしてきた
 - spack dev-build など
- 新規Spack機能により、開発者用環境が使用可能に
 - コードで作業可能
 - 依存関係を保ったままの複数のパッケージの開発
 - 変更時リビルドが簡単
- spack 環境上でビルド
 - 開発パッケージのインストールモデルに必要な変更
 - 開発パッケージは設定変更でパスを変更しない
 - 開発者はビルドを素早く反復可能

```
$ spack env activate .
$ spack add myapplication
$ spack develop axom@0.4.0
$ spack develop mfem@4.2.0

$ ls
spack.yaml  axom/  mfem/

$ cat spack.yaml
spack:
  specs:
    - myapplication      # depends on axom, mfem

  develop:
    - axom @0.4.0
    - mfem @develop
```

SpackはAMLチームの開発環境合理化に貢献

• Spack導入前

- 誰もが独自で Python/PyTorchをゼロから構築していた
- スクリプトを書き、渡していた
- スクリプトはゆっくりと変更とマジックを蓄積
- ビルドの違いをデバッグするのに何日も費やしていた

• Spack導入後

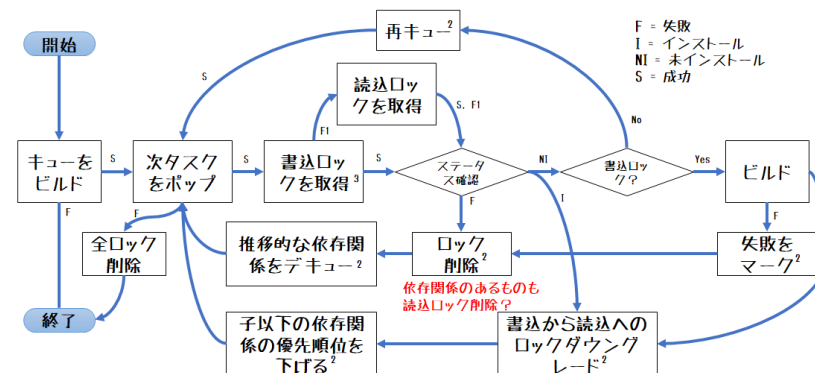
- リポジトリにてバージョン管理された再現可能なSpack環境
- 共有チームディレクトリ内の標準環境
- **どんなチームメンバーでも20分以内にカスタマイズ済み実行環境を得られるように**
 - Pythonバージョンの変更、PyTorchバージョンの変更など

```
spack:
  specs:
    - py-horovod
    - py-torch
    - python
    - py-h5py
  packages:
    all:
      providers:
        mpi:
          - mvapich2@2.3
        lapack:
          - openblas threads=openmp
        blas:
          - openblas threadd=openmp
      buildable: true
      variants: [+cuda cuda_arch=37]
      compiler: [gcc@7.3.0]
  python:
    version: [3.8.6]
  cudnn:
    version:
      - 8.0.4.30-11.1-linux-x64
  py-torch:
    buildable: true
    variants: +cuda +distributed
  mvapich2:
    externals:
      - spec: mvapich2@2.3.1%gcc@7.3.0
        prefix: /usr/tce/packages/mvapich2/mvapich2-2.3-gcc-7.3.0
  compilers:
    - compiler:
        operating_system: rhel7
        paths:
          cc: /usr/tce/packages/gcc/gcc-7.3.0/bin/gcc
          cxx: /usr/tce/packages/gcc/gcc-7.3.0/bin/g++
```

Configure and build complex software stacks with a single spack.yaml file

```
srun -N -n 8 spack install .
```

- ## 分散ロックングアルゴリズム



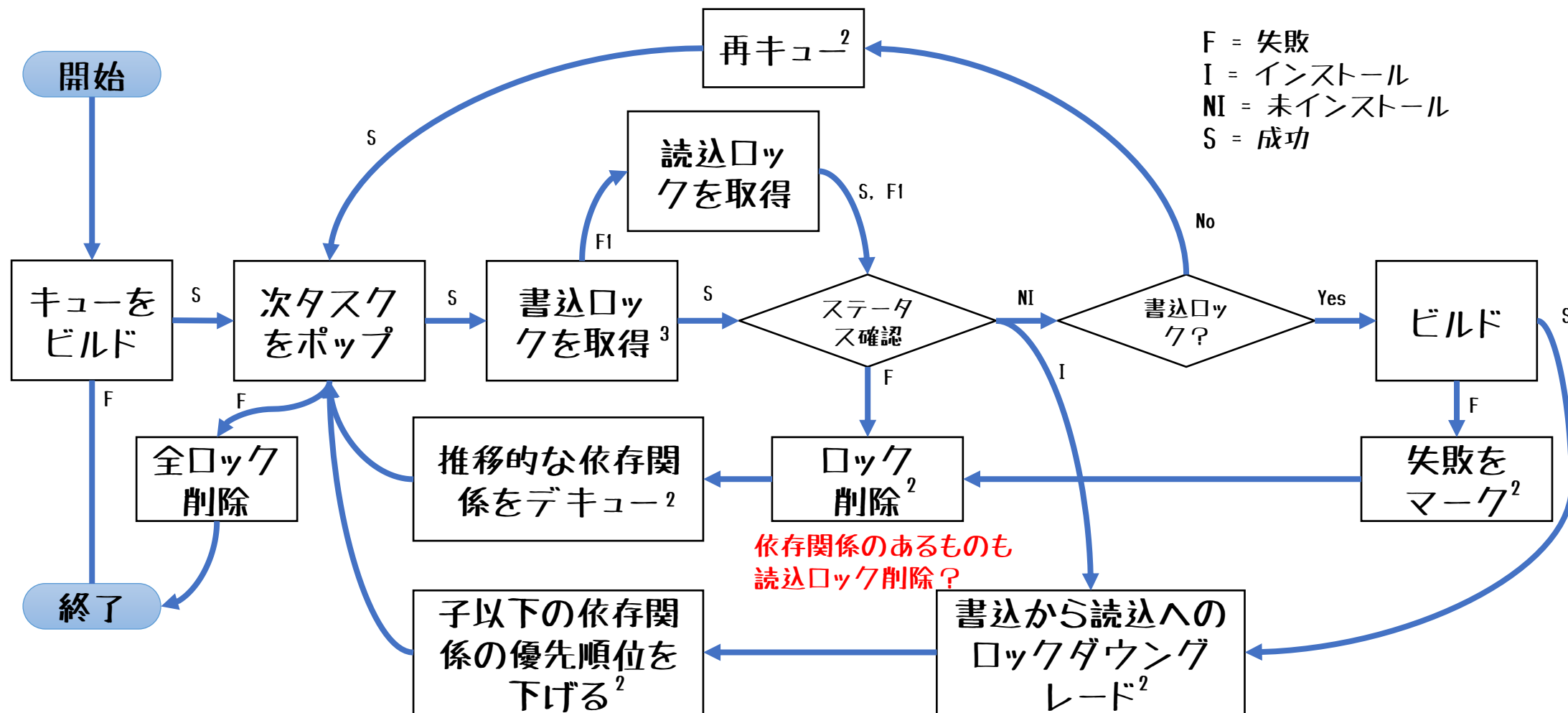
```

spack:
  spec:
    - openmpi@4.1.0
    - py-libcsm@python3.7.3
    - hpcx
    - mpmc
    - trillinos@12.14.1+ath+intrepid2+shards
    - sundials
    - strumpack
    - superlu-dist
    - superlu
    - tasmanian
    - mercury
    - mercury
    - hdl
    - adios2
    - ginkgo
    - pdt
    - tau
    - hpcxtoolkit
  packages:
    all:
      provider:
        mpix [spectrum-mpi]
        target: [ppc64le]
      cuda:
        buildable: false
        version: [10.3.242]
      modules:
        cuda@10.3.242: cuda@10.3.242
  spectrum-mpi:
    buildable: false
    version:
      - 10.3.3.2
    modules:
      spectrum-mpi@10.3.3.2: spectrum-mpi/10.3.3.2-20200722
  config:
    misc_cache: $spack/cache
    build_stage: $spack/build-stage
    install_tree: $spack/spackinstall
  view: false
  concretization: separately

```

E4S マニフェスト

分散ロックングアルゴリズム



ビルド構成がそれ自体の多次元制約最適化問題である

- v0.16.0における新しい concretizer によりこの問題を解く事が可能
 - NP困難最適化問題を解くためのフレームワーク 回答セットプログラミングを使用
 - 他のシステムとは異なり、パッケージマネージャはビルド詳細や構成への洞察が可能
- ASPプログラムは2つのパートで構成される：
 - 大規模ファクトリスト
 - パッケージリポジトリから生成される
 - 20,000~30,000 のファクトは典型的
 - includeの依存関係、バージョン、オプションほか
 - 小さなロジックプログラム
 - 800行以下のASPコード
 - 300のルール + 11の最適化基準

```
%-----  
% Package: ucx  
%-----  
version_declared("ucx", "1.6.1", 0).  
version_declared("ucx", "1.6.0", 1).  
version_declared("ucx", "1.5.2", 2).  
version_declared("ucx", "1.5.1", 3).  
version_declared("ucx", "1.5.0", 4).  
version_declared("ucx", "1.4.0", 5).  
version_declared("ucx", "1.3.1", 6).  
version_declared("ucx", "1.3.0", 7).  
version_declared("ucx", "1.2.2", 8).  
version_declared("ucx", "1.2.1", 9).  
version_declared("ucx", "1.2.0", 10).  
  
variant("ucx", "thread_multiple").  
variant_single_value("ucx", "thread_multiple").  
variant_default_value("ucx", "thread_multiple", "False").  
variant_possible_value("ucx", "thread_multiple", "False").  
variant_possible_value("ucx", "thread_multiple", "True").  
  
declared_dependency("ucx", "numactl", "build").  
declared_dependency("ucx", "numactl", "link").  
node("numactl") :- depends_on("ucx", "numactl"), node("ucx").  
  
declared_dependency("ucx", "rdma-core", "build").  
declared_dependency("ucx", "rdma-core", "link").  
node("rdma-core") :- depends_on("ucx", "rdma-core"), node("ucx").  
  
%-----  
% Package: util-linux  
%-----  
version_declared("util-linux", "2.29.2", 0).  
version_declared("util-linux", "2.29.1", 1).  
version_declared("util-linux", "2.25", 2).  
  
variant("util-linux", "libuuid").  
variant_single_value("util-linux", "libuuid").  
variant_default_value("util-linux", "libuuid", "True").  
variant_possible_value("util-linux", "libuuid", "False").  
variant_possible_value("util-linux", "libuuid", "True").  
  
declared_dependency("util-linux", "pkgconfig", "build").  
declared_dependency("util-linux", "pkgconfig", "link").  
node("pkgconfig") :- depends_on("util-linux", "pkgconfig"), node("util-linux").  
  
declared_dependency("util-linux", "python", "build").  
declared_dependency("util-linux", "python", "link").  
node("python") :- depends_on("util-linux", "python"), node("util-linux").
```

SpackソルバーのASP入力サンプル

新規 concretizer により、パッケージ、特に SDK の複雑な制約の大幅な簡素化が可能に

- SDK 内の依存関係やその他制約は非常に厄介になる場合がある

場合によって、依存関係オプションのクロスプロダクトが必要だった：

導入前

```
depends_on( 'foo+A+B' , when= 'a+b' )  
depends_on( 'foo+A~B' , when= 'a~b' )  
depends_on( 'foo~A+B' , when= '~a+b' )  
depends_on( 'foo~A~B' , when= '~a~b' )
```

- 新しい concretizer は、苦痛となっていた構成の一部を取り除く

導入後

```
depends_on( 'foo' )  
depends_on( 'foo+A' , when= 'a' )  
depends_on( 'foo+B' , when= 'b' )
```

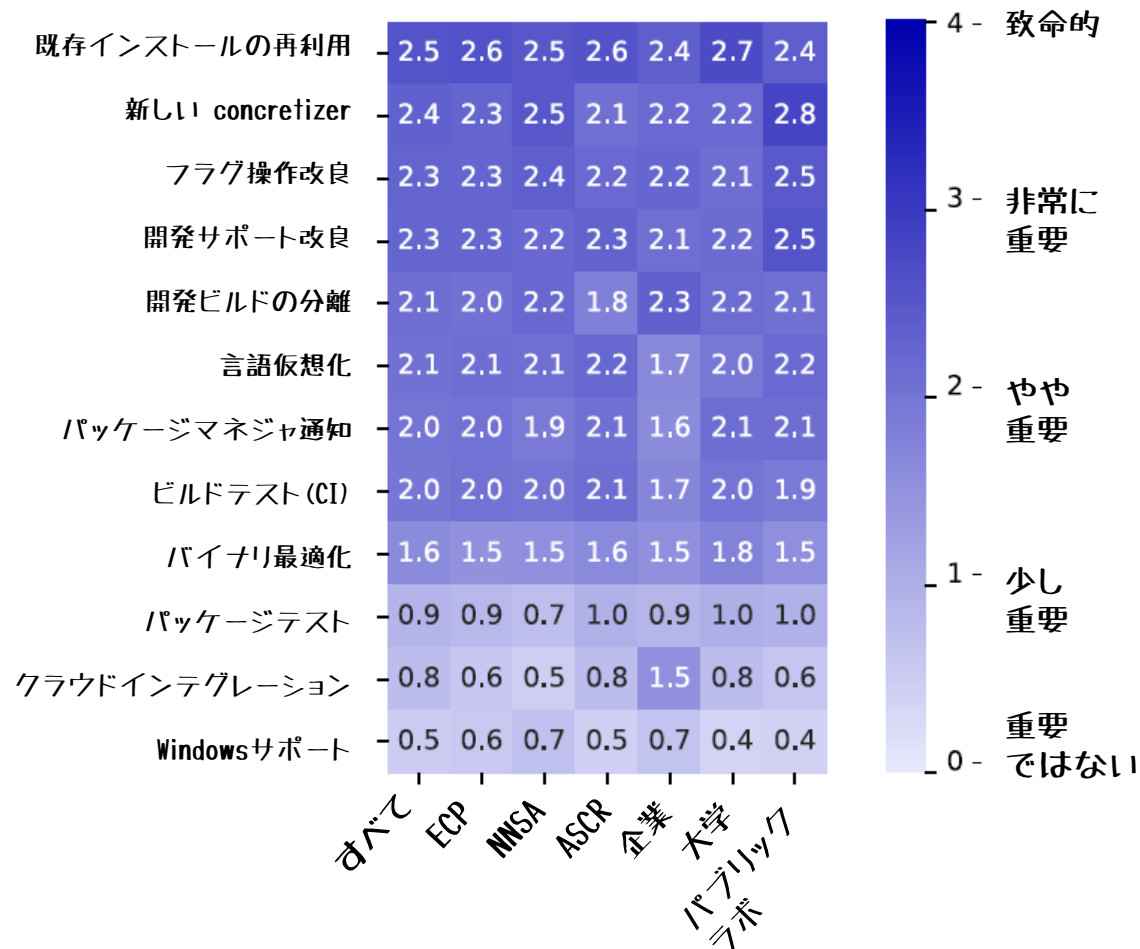
- 依存関係に特化して扱うなどの新規構成も可能

- 条件がより一般化された場合
- 他の制約と一緒に解決が可能

仮想特化はいこれまで機能しなかった：

```
depends_on( 'foo+A+B' , when= 'a+b' )  
depends_on( 'blas' )  
depends_on( 'openblas threads=openmp' , when= '^openblas' )
```


Spackに求められる機能のトップ6のうちの4つは、新しい concretizer に関連

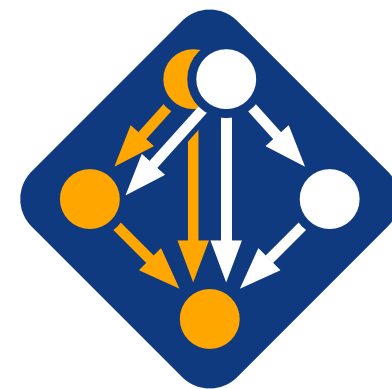


- Spackパッケージの複雑さが増えている
 - さらに多くのパッケージ解決には1年以上のトライ & エラーが必要
 - 多くのバリエーション、条件付き依存関係、特定のコンパイラ要件
- 既存インストールをより積極的に活用するには、依存関係のより適切な解決が必要となる
 - 既存インストールパッケージでビルドを構成する方法を分析できる必要がある
- ビルドの依存関係を個別に解決するには、より高度なソルバーも必要となる
 - さらに多くの組み合わせで解決
 - コンパイラの混合をサポートする必要があるため、異なるパッケージのビルド要件間でバージョンが競合する

v0.17がまもなくリリース

主なゴール

1. 旧 concretizer を削除し、新しい concretizer をデフォルトに
2. バイナリキャッシュワークフローを改善・強化
3. 既存インストールパッケージおよびバイナリミラーの再利用を Spack で最適化可能に
4. “共有” spack インスタンス管理を容易に
5. ~/.spack 構造のような問題点の除去

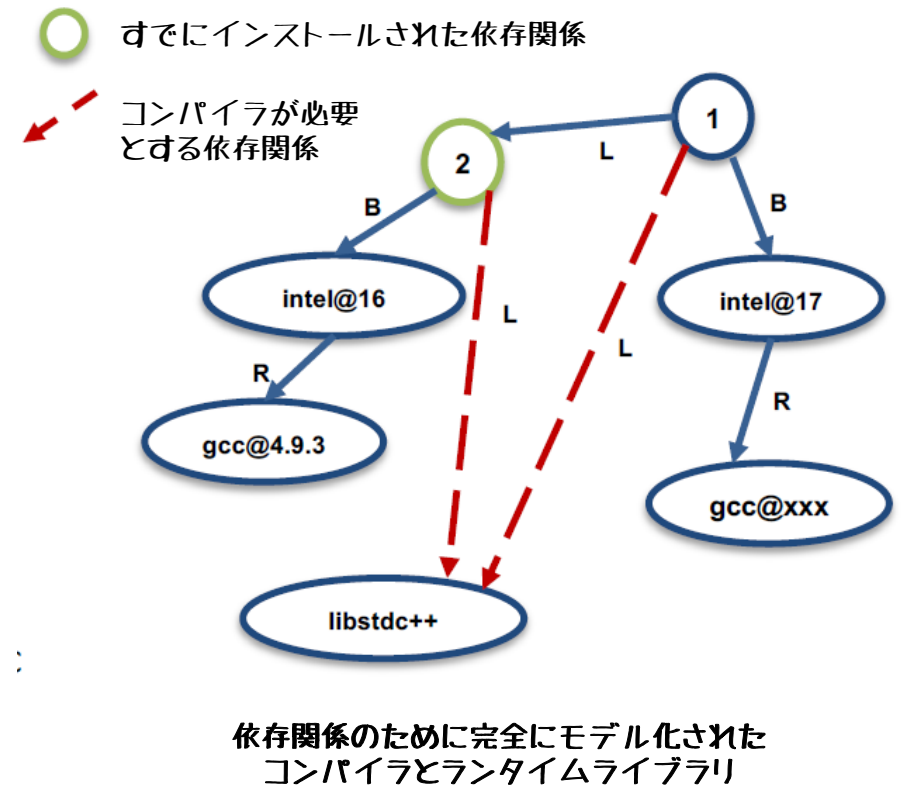


Spack0.17ロードマップ：権限とディレクトリ構造

- Spackインスタンスの共有
 - 多くのユーザはクラスタにSpackをインストールし、`module load spack` を実行したいと考えている
 - Spackプレフィックス内へのインストレーションをユーザ間で共有
 - デフォルトではSpackはホームディレクトリにインストール
 - 共有するにはほとんどの状態出力をSpackプレフィックスから移動する必要がある
 - インストレーションは `~/.spack/...` に移動される
- `~/.spack` ディレクトリ構造の削除
 - インストレーションをホームディレクトリに移動するため、構造に問題が発生
 - ユーザ構造は、望まないグローバルのようなもの(例: `LD_LIBRARY_PATH` 😬)
 - CIビルドを妨害(多くのユーザは `rm -rf ~/spack` を使って回避)
 - 再現性確保のための努力に背く
 - 複数マシン間で構成を管理するのは困難
 - 環境は、はるかに適している
 - 単一の構造ではなく、環境内でユーザがこのような環境を維持できるようにする

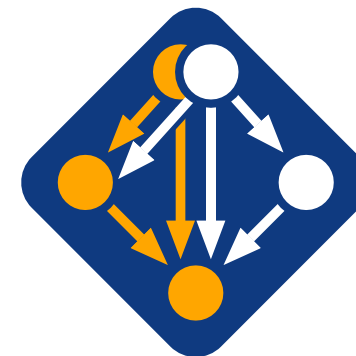
Spack0.18ロードマップ：依存関係としてのコンパイラ

- コンパイラの相互互換性に対応するにはより深いモデリングが必要
 - libstdc++、libc++の互換性
 - コンパイラに依存するコンパイラ
 - 複数のコンパイラを使って実行可能ファイルをリンク
- 最初のプロトタイプが完成！
 - コンパイラ依存関係を使っていくつかのパッケージのビルドに成功
 - 発展するには新しい concretizer が必要！
- 言語依存パッケージ
 - cxx@2011、cxx@2017、fortran@1955などに依存
 - [openmp@4.5](#)やその他コンパイラ機能に依存
 - 言語、openmp、cudaなどを仮想としてモデル化



Spackコミュニティに参加しよう！

- 参加方法は色々
- github.com/spack/spack にてパッケージ、ドキュメント、機能を提供
- github.com/spack/spack に構成を提供
- 我々と対話を！
- Slackチャンネル (spackpm.herokuapp.com) にはすでに参加済み
- Google グループに参加 (GitHub リポジトリの情報を参照のこと)
- GitHub に Issue を送信し、pull request！



★ Star us on GitHub!
github.com/spack/spack



Follow us on Twitter!
[@spackpm](https://twitter.com/spackpm)

HPCソフトウェアの配布・利用が簡単になることを願って



免責事項

この文書は、米国政府の機関が後援した作業の説明として作成されました。米国政府もLawrence Livermore National Security, LLCもその従業員も、明示または黙示を問わず、いかなる情報、装置、製品、または開示されたプロセス、またはその使用が個人所有の権利を侵害しないことを表す。商号、商標、製造元、またはその他の方法による特定の商用製品、プロセス、またはサービスへの言及は、必ずしも米国政府またはローレンスリバモアナショナルセキュリティLLCによるその承認、推奨、または支持を構成または暗示するものではありません。

本書に記載されている著者の見解および意見は、必ずしも米国政府またはローレンスリバモアナショナルセキュリティLLCの見解および意見を表明または反映するものではなく、広告または製品の推奨目的で使用されるものではありません。